UNIVERSITATEA "POLITEHNICA" TIMIȘOARA

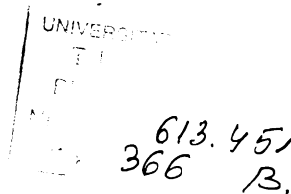
Facultatea de Automatică şi Calculatoare

Contribuţii la proiectarea sistemelor de operare distribuite

Teză de doctorat

Autor: Ing. Carmen Muşatescu
Conducător ştiinţific: Prof. univ. dr. ing. Ioan Jurcă

BIBLIOTECA CENTRALĂ
UNIVERSITATEA "POLITEHNICA"
TIMIŞOARA



Timişoara
1997

BUPT

62-52
681.3.066

BUPT

PREFAŢĂ

Lucrarea de faţă se înscrie în tendinţele actuale de migrare de la prelucrarea centralizată la prelucrarea distribuită, migrare care presupune modificarea modului de proiectare a aplicaţiilor, modificarea fluxului prelucrării şi controlului datelor precum şi adaptarea structurilor organizatorice la noile structuri informaţionale, mai ieftine şi mai eficiente. Prelucrarea distribuită îşi propune să pună în valoare, într-un mod cît mai eficient, avantajele descentralizării, astfel încît să răspundă unor noi cerinţe ale utilizatorilor, printre care se enumeră: (1) instrumente sosftware pentru comunicare şi partajare a informaţiilor; (2) luarea deciziilor într-o manieră cooperativă; (3) disponibilitatea prmanentă a serviciilor; (3) posibilitatea modificării progresive a echipamentelor hardware şi a modulelor software pentru a putea urmări evoluţia, foarte rapidă în ultimul domeniu deceniu, a tehnologiei.

O mare parte din aria prelucrării distribuite este acoperită de reţelele locale de date. Ele au cunoscut şi continuă să cunoască o dezvoltare extrem de puternică, materializată atît în numărul de reţele locale existente cît şi în numărul de aplicaţii efectiv implementate. Se apreciază că acesta este, probabil, domeniul cel mai dinamic al sistemelor de prelucrare distribuită.

Obiectivul lucrării de faţă se înscrie pe linia proiectării unor subsisteme software care să poată fi utile în proiectarea de aplicaţii distribuite în reţele locale. Implementarea acetor suporturi a fost realizată în sistemele de operare Unix, Microsoft Windows - utilizîndu-se funcţii Win 32, implementările sunt valabile atît în Windows NT cît şi în Windows 95 - şi în Novell NetWare.

Lucrarea reprezintă rezultatul activităţii desfăşurate de autor pe parcursul a 7 ani, în cadrul Catedrei de Calculatoare a Facultăţii de Automatică, Calculatoare şi Electronică, Universitatea din Craiova. În cadrul lucrării, am abordat atît aspectele teoretice şi problematica sistemelor de operare distribuite cît şi aspectele practice legate de concepţia, proiectarea şi implementarea unor subsisteme ale sistemelor de operare distribuite, care să asiste utilizatorul în dezvoltarea de aplicaţii distribuite.

Pe toată durata eleborării tezei, am beneficiat de îndrumarea atentă, exigentă şi competentă a conducătorului ştiinţific, prof. dr. ing. Ioan Jurcă, care, cu generozitate şi înţelegere mi-a conturat domeniul pe care l-am abordat în activitatea de doctorat. Pentru sprijinul profesional şi moral, pentru amabilitatea deosebită pe care mi-a acordat-o şi nu în ultimul rînd pentru bibliografia pe care mi-a pus-o la dispoziţie, pe tot parcursul elaborării lucrării, ţin sa-mi exprim întreaga recunoştinţă, consideraţie şi stimă, alături de cele mai respectuoase mulţumiri. Acest lucru are o rezonanţă mai profundă întrucît apreciez că prof. dr. ing. Jurcă şi-a adus o contribuţie esenţială în formarea personalităţii mele în calitate de cadru didactic la Catedra de Calculatoare, Facultatea de Automatică, Calculatoare şi Electronică, Universitatea din Craiova.

Aduc, de asemenea, mulţumiri întregii Catedre de Calculatoare a Facultăţii de Automatică şi Calculatoare, Universitatea "Politehnica" Timişoara pentru modul în care m-a primit în mijlocul său pe parcursul activităţii de doctorat, pentru încurajările susţinute pe care mi le-a adresat.

Datorez mulţumiri şi Colectivului Catedrei de Calculatoare, Facultatea de Automatică, Calculatoare şi Electronică, Universitatea din Craiova pentru răbdarea de care a dat dovadă în situaţiile în care am "monopolizat" uneori reţelele locale aflate în dotare.

Autorul,

BUPT

CUPRINS

INTRODUCERE.. intro -1

PARTEA I: SISTEME DE OPERARE DISTRIBUITE

CAPITOLUL 1. SISTEME DE OPERARE DISTRIBUITE. SUPORTUL OFERIT PRELUCRĂRILOR DISTRIBUITE, PROBLEME SPECIFICE, MODELE DE PROIECTARE 1
1.1. Introducere.. 2
1.2. Prelucrarea distribuită a datelor.. 2
1.2.1. Obiectivele sistemelor distribuite.. 2
1.2.2. Modele actuale în prelucrarea distribuită.. 3
1.2.3. Interacţiunea prelucrare distribuită - sistem de operare 4
1.2.3.1. Proiectarea unui suport pentru conectarea aplicaţiilor în mediu distribuit ... 5
1.2.3.2. OSF DCE - Strategie şi arhitectură.. 8
1.3. Sisteme de operare pentru reţele de calculatoare şi sisteme de operare distribuite......... 10
1.3.1. Specificul sistemelor de operare pentru reţele de calculatoare 11
1.3.1.1. Limbajul de comandă.. 11
1.3.1.2. Procesele agent.. 12
1.3.2. Specificul sistemelor de operare distribuită.. 13
1.3.2.1. Gestionarea proceselor 13
1.3.2.2. Gestiunea fişierelor 18
1.4. Modele în proiectarea sistemelor de operare distribuite 24
1.4.1. Modelul proces 24
1.4.2. Modelul obiect în proiectarea sistemelor de operare distribuite.. 25

CAPITOLUL 2. COMUNICAŢII ÎN SISTEME DISTRIBUITE. PROTOCOALE. 27
2.1. Conceptul de arhitectura deschisă. Stratificarea ierarhică a protocoalelor.................... 27
2.1.1. Modelul arhitectural ISO/OSI.. 28
2.1.2. Nivele dependente de reţea. Prezentare generală.. 31
2.1.3 Nivelul transport, principalul furnizor de servicii pentru sistemele de operare....... 38
2.1.4. Nivele suport pentru aplicaţii.. 47
2.1.5. Solutii de implementare a modelului OSI.. 49
2.1.6. Importanţa modelului OSI pentru sisteme distribuite.. 50
2.2. Exemple de protocoale.. 51
2.2.1. Familia de protocoale TCP/IP.. 51
A. Protocoale IP. Noua generaţie de protocol IP, versiunea 6.. 52
B. Protocolul TCP.. 55
C. Comunicarea prin socluri în reţele UNIX şi Microsoft Windows (Win32).......... 57
2.2.2. Familia de protocoale Novell IPX/SPX 60
2.3. Aspecte specifice ale suportului de comunicaţie pentru sistemele de operare distribuite.. 61
2.4. Protocoale cu caracter universal şi protocoale cu caracter restrîns.. 63

BUPT

CAPITOLUL 3. PARADIGME ALE INTERACȚIUNII DINTRE PROCESE ÎN SISTEME DISTRIBUITE 64
3.1. Modelul client-server ... 65
3.1.1. Prezentare generală a modelului client/server ... 65
3.1.2. Concurența la nivelul serverelor şi clienților ... 67
3.1.3. Protocoale pentru modelul client-server ... 73
3.1.4. Sisteme client-server ... 75
3.2. Comunicarea prin mesaje ... 77
3.2.1. Probleme specifice ale comunicării prin mesaje ... 77
3.2.2. Primitive cu şi fără blocare ... 78
3.2.3 Primitive cu şi fără bufferare ... 79
3.2.4. Primitive sigure şi nesigure ... 80
3.2.5 Forme structurate ale comunicării prin mesaje ... 81
3.2.6. Adresare directă şi indirectă ... 81
3.2.7. Excepții ... 83
3.2.8 Exemplu. Comunicația prin porturi în Mach ... 83
3.3. Comunicarea între procese prin conducte ... 85
3.3.1. Prezentarea generală a comunicației prin conducte ... 85
3.3.2 Comunicarea prin conducte în Microsoft WIN32 ... 85
3.4. Modelul apelului de procedură la distanță ... 86
3.4.1. Prezentarea generală a modelului ... 87
3.4.2. Transmiterea parametrilor şi rezultatelor ... 88
3.4.3. Legarea clienților la servere ... 89
3.4.4. Probleme de concurență ... 91
3.4.5 Semantica RPC în prezența defectelor ... 91
3.4.6. Protocoale pentru RPC ... 92
3.4.7. Extensii ale mecanismului de bază RPC ... 93
3.4.8 Aspecte critice în proiectarea comunicării prin RPC ... 94
3.4.9. Exemple ... 96
A. Modelul RPC în DCE ... 96
B. Microsoft RPC ... 98
3.5. Comunicații de grup ... 101
3.5.1. Probleme specifice de comunicații de grup ... 101
3.5.2. Primitive şi protocoale pentru comunicații de grup ... 101
3.5.3. Exemple: MCL, ISIS, Amoeba ... 102

CAPITOLUL 4. SINCRONIZAREA ÎN SISTEME DE OPERARE DISTRIBUITE 105
4. 1. Ordonarea evenimentelor în sisteme distribuite ... 105
4.2. Soluții centralizate pentru sincronizare ... 106
4.2.1. Utilizarea unui ceas fizic global ... 106
4.2.2. Utilizarea unui proces central ... 106
4.2.3. Contori de evenimente şi secvențiatori ... 107
4.3. Soluții distribuite pentru sincronizare ... 109
4.3.1. Sincronizarea ceasurilor ... 109
4.3.2. Algoritmi distribuiți pentru excludere reciprocă ... 112
4.3.2.1. Algoritmul lui Lamport ... 112
4.3.2.2. Algoritmul lui Ricard şi Agrawala ... 114
4.3.2.3. Algoritm pentru excludere reciprocă în accesul la K resurse identice ... 116
4.3.2.4. Alți algoritmi ... 118
4.3.2.5. Considerații finale ... 118

BUPT

4.3.2.6. Implementarea algoritmilor distribuiţi pentru excludere reciprocă în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare 119
4.3.3. Algoritmi care utilizează un token circulant pentru excludere reciprocă 120
4.3.3.1. Sisteme structurate în inel 120
4.3.3.2. Algoritmul pentru sisteme nestructurate în inel 122
4.3.3.3. Secvenţiator circulant 123
4.3.3.4. Implementarea algoritmilor care utilizează un token circulant în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare 124
4.3.4. Algoritmi broadcast 125
4.3.4.1. Semafoare distribuite 125
4.3.4.2. Contori de evenimente distribuiti 127
4.3.4.3. Secvenţiatori distribuiţi 129
4.3.4.3. Implementarea algoritmilor broadcast în reţele Unix şi Microsoft Windows (Win32) 130
4.4. Algoritmi de alegere 131
4.4.1. Algoritmul Bully 131
4.4.2 Algoritm în inel 132

CAPITOLUL 5. PROIECTAREA ŞI IMPLEMENTAREA UNOR MECANISME SUPORT PENTRU DEZVOLTAREA DE APLICAŢII DISTRIBUITE ÎN REŢELE UNIX, MICROSOFT WINDOWS (WIN32) ŞI NOVELL NETWARE 133
5.1. Proiectarea unui toolkit pentru controlul accesului concurent la o bază de date în reţele locale Unix, Windows (Win32) sau Novell NetWare 133
5.1.1. Soluţii bazate pe semafoare 133
A. Prima soluţie bazată pe semafoare 134
B. A doua soluţie cu semafoare 135
C. A treia soluţie cu semafoare 136
5.1.2. Soluţie bazată pe un proces coordonator 137
5.1.3. Implementări în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare 139
5.1.3.1. Implementarea soluţiilor bazate pe semafoare în reţele Novell Netware 139
5.1.3.2. Implementarea soluţiilor bazate pe semafoare în reţele Unix şi Windows (Win32) 139
5.1.3.3. Implementarea soluţiei bazată pe un proces coordonator în reţele Novell Netware 140
5.1.4. Compararea soluţiilor propuse şi concluzii 140
5.2. Proiectarea şi implementarea unui toolkit pentru comunicaţii de grup 141
5.2.1. Cerinţe de proiectare 141
5.2.2. Varianta semidistribuită 145
5.2.2.1. Arhitectura generală a sistemului TGC 145
5.2.2.2. Protocolul de comunicaţie 146
5.2.2.3. Descrierea serverelor locale şi a serverului central 147
5.2.3. Alte variante 159
5.2.3.1. Variante complet distribuite 159
A. Varianta în care se foloseste transmisia broadcast 159
B. Varianta în care nu se foloseste transmisie broadcast 160
5.2.3.2. Varianta în inel 161
5.2.4. Implementarea variantelor complet distribuite 161
5.2.4.1. Implementarea variantei cu transmisie broadcast 161
5.2.4.2. Implementarea variantei cu transmisie în inel 166
5.2.5. Comparatie intre variantele propuse şi concluzii 171

BUPT

PARTEA a II a: CRITERII DE PROIECTARE A SISTEMELOR DE PROGRAMARE DISTRIBUITĂ BAZATĂ PE OBIECTE (SPDBO). PROIECTAREA UNUI SUPORT SOFTWARE BAZAT PE MODELUL TRANZACŢIILOR ATOMICE PENTRU SPDBO ÎN REŢELE UNIX ŞI WINDOWS

CAPITOLUL 6. SISTEME DE PROGRAMARE DISTRIBUITĂ BAZATE PE OBIECTE (SPDBO). CRITERII DE PROIECTARE 161

6.1. Concepte fundamentale........ 161
6.2. Structura obiectelor........ 162
6.3. Gestionarea obiectelor........ 164
6.3.1. Gestionarea acţiunilor........ 164
6.3.2. Sincronizarea acţiunilor........ 164
6.3.3. Siguranţa obiectelor........ 165
6.4. Interacţiunea obiectelor........ 166
6.4.1. Localizarea obiectelor........ 166
6.4.2. Apelul de operaţii........ 166
6.4.3. Detectarea eşecurilor în apelarea operaţiilor........ 167
6.5. Reprezentarea obiectelor în memorie........ 168

CAPITOLUL 7. CONTROLUL CONCURENŢEI ŞI RECUPERAEA OBIECTELOR 169

7.1 Introducere........ 169
7.2 Modelul tranzacţional........ 170
7.2.1 Consistenţa obiectelor. Noţiunea de tranzacţie........ 170
7.2.2 Modelul de lucru pentru sistemul distribuit de obiecte........ 171
7.2.3 Cerinţele modelului sistemului distribuit de obiecte........ 173
7.3 Realizarea atomicităţii tranzacţiilor........ 173
7.3.1 Protocoale pentru comiterea tranzacţiilor........ 173
7.3.2 Protocolul CCR în OSI........ 175
7.3.3. Implementarea tranzacţiilor atomice........ 176
7.3.4. Recuperarea obiectelor........ 177
7.3.4.1. Protocolul Write-Ahead Logging........ 178
7.3.4.2. Tehnicile Value-Logging şi Operation Logging........ 178
7.3.4.3. Tehnicile Redo-Only Logging şi Undo-Only Logging........ 179
7.3.4.4. Structura sistemului de recuperare........ 179
7.4. Controlul concurenţei........ 180
7.4.1. Cerinţele mecanismului de control al concurenţei........ 180
7.4.2 Serializarea tranzacţiilor........ 180
7.4.3. Mecanisme tradiţionale pentru controlul concurenţei........ 184
7.4.4. Mecanisme bazate pe blocarea cu lock-uri........ 184
7.4.4.1. Mecanismul blocării în două faze (2PL)........ 185
7.4.4.2 Algoritmi de blocare ierarhici........ 186
7.4.4.3 Grafuri aciclice de lock-uri........ 188
7.4.5 Mecanisme bazate pe marca de timp........ 188
7.4.5.1 Concepte de bază........ 188
7.4.5.2 Algoritmul fundamental pentru controlul concurenţei........ 189
7.4.5.3. Optimizări ale algoritmului de bază........ 190
7.4.5.4. Algoritmul conservativ de ordonare a mărcilor de timp........ 191
7.4.6 Controlul optimist al concurenţei........ 192

BUPT

7.4.6.1. Concepte de baza 192
7.4.6.2. Algoritmi pentru controlul optimist al concurenței 192
7.4.7 Extinderea mecanismelor tradiționale pentru controlul concurenței 194
7.4.7.1 Cerințe ale controlului concurenței în aplicațiile avansate 194
7.4.7.2 Extinderea mecanismelor bazate pe serializabilitate și renunțarea la serializabilitate 195
7.4.8. Mecanisme pentru controlul concurenței în aplicații avansate 195
7.4.8.1 Mecanismul blocării altruiste 195
7.4.8.2 Mecanismul validării instantanee 197
7.4.9. Mecanisme pentru controlul concurenței bazate pe informații semantice. Sincronizarea accesului la tipuri de date abstracte 98
7.4.9.1. Relația de dependență 198
7.4.9.2. Specificarea tipurilor abstracte 200
7.4.9.3. Metode de sincronizare 205

CAPITOLUL 8. CONTROLUL INTERBLOCĂRII OBIECTELOR **208**
8.1. Metode de control al interblocării 208
8.2. Prevenirea interblocărilor 208
8.3. Detecția interblocărilor în sisteme distribuite 209
8.3.1 Modelul de lucru 209
8..3.2 Clase de algoritmi pentru detecția interblocării 210
8..3.2.1.Clasificarea algoritmilor 210
8..3.2.2 Modele de interblocare 210
8.3.2.3 Clase de algoritmi distribuiti pentru detecția interblocării 211
8.3.3 Algoritmi centralizați pentru detecția interblocării 211
8.3.4. Algoritmul ierarhic pentru detecția interblocării 212
8.3.5. Algoritmi distribuiți pentru detecția interblocării 212
8.3.5.1. Algoritm edge-chasing pentru detecția interblocării în modele AND 212
8.3.5.2 Algoritm cu difuzie pentru detecția interblocării în modele OR 213

CAPITOLUL 9. PROIECTAREA UNUI SUPORT SOFTWARE PENTRU PROGRAMARE DISTRIBUITĂ BAZATĂ PE OBIECTE SI PE TRANZACȚII ATOMICE **214**
9.1. Cerințe de proiectare 214
9.2. Modelul sistemelor DOSTP 217
9.3. Arhitectura generală a sistemului DOSTP 218
9.3.1. Procesele utilizator ale sistemului DOSTP 219
9.3.2. Managerul tranzacțiilor 219
9.3.3. Serverele de comunicații 219
9.3.4. Serverele de nume 220
9.3.5. Managerul central 220
9.3.6. Managerii de obiecte 220
9.3.7. Sistemul de recuperare 220
9.4. Proiectarea elementelor componente ale sistemului DOSTP 221
9.4.1. Structuri de date ale sistemului 221
9.4.2. Integrarea proceselor de aplicație în sistemul DOSTP 224
9.4.3. Proiectarea managerilor de tranzactii 227
9.4.4. Proiectarea serverelor de comunicații 234
9.4.5. Proiectarea serverelor de nume 239
9.4.6. Proiectarea managerului central 240
9.4.7. Proiectarea managerilor de obiecte 241

BUPT

CAPITOLUL 10. IMPLEMENTAREA SISTEMULUI DOSTP ÎN REȚELE UNIX ȘI MICROSOFT WINDOWS (WIN32) 244
10.1. Particularități de implementare........ 244
10.1.1. Particularități de implementare în rețele Unix........ 244
10.1.2. Particularități de implementare în rețele Microsoft Windows (Win 32)........ 246
10. 2. Exemple de desfășurare a comunicațiilor în sistem........ 247
10.2.1.Conectarea nodurilor și a managerilor de obiecte la sistemul tranzacțional........ 247
10.2.2. Inițierea unei tranzacții........ 248
10.2.3. Exemple de implicare a obiectelor în tranzacții........ 248
10.2.4. Exemple de desfășurare a protocolului two-phase commit........ 250
10.2.5. Exemple de tratare a situațiilor de recuperare........ 251
10.3. Exemplu de implementare al unui manager de obiecte........ 253
10.3.1. Descrierea managerului de obiecte........ 253
10.3.1.1. Organizarea managerului de obiecte........ 254
10.3.1.2. Concurența la nivelul managerului de obiecte........ 255
10.3.1.2. Recuperarea obiectelor........ 257
10.3.2. Conectarea obiectelor în sistemul DOSTP........ 258
10.3.2.1. Descrierea clasei CObjectManager........ 258
10.3.2.2. Descrierea clasei CDispatcher........ 258
10.3.2.3. Integrarea automată a obiectelor în sistemul tranzacțional........ 258
10.3.3. Particularități ale implementării în sistemele Unix și Windows........ 260
10.3.4. Exemplu de aplicație care apelează managerii de obiete de tipul proiectat........ 261
10.4. Concluzii și direcții de dezvoltare ulterioară........ 262

CAPITOLUL 11. CONCLUZII GENERALE ȘI CONTRIBUȚII ORIGINALE 263
11.1. Concluzii generale........ 263
11.2. Contribuții originale, valoare aplicativă și direcții de dezvoltare ulterioară........ 269

ANEXE

ANEXA A. IMPLEMENTAREA UNUI TOOLKIT PENTRU COMUNICAȚII DE GRUP. MANAGERUL DE GRUPURI PENTRU VARIANTELE DISTRIBUITE anexaA-1

ANEXA B. SINCRONIZARE ÎN SISTEME DISTRIBUITE
B1. Exemple de programe client care, pentru realizarea excluderii reciproce în rețele locale, folosesc un server care implementează unul din algoritmii prezentați în capitolul 4, la 4.3.2 și 4.3.3........ anexaB-1
B2. Implementarea algoritmului Lamport pentru obținerea excluderii reciproce - varianta Unix........ anexaB-4
B3. Implementarea algoritmului Ricard si Agrawala pentru obținerea excluderii reciproce varianta Novell NetWare........ anexaB-7
B4. Implementarea algoritmului pentru obținerea excluderii reciproce în accesul la k resurse -varianta Windows........ anexaB-9
B5. Implementarea algoritmului bazat pe un token circulant pentru obținerea excluderii reciproce- varianta Novell NetWare........ anexaB-11
B6. Implementarea algoritmului Chandy pentru obținerea excluderii reciproce -varianta Unix........ anexaB-12
B7. Implementarea semafoarelor distribuite -varianta Windows........ anexaB-15
B8. Implementarea contorilor de eveniment distribuiți -varianta UNIX........ anexaB-18

BUPT

B9. Implementarea secvenţiatorilor distribuiţi -varianta UNIX.................................. anexaB-20
B10. Implementarea excluderii reciproce folosind secvenţiatori si contori de eveniment distribuiţi- varianta Unix.. anexaB-24

ANEXA C. PROIECTAREA UNUI TOOLKIT PENTRU CONTROLUL ACCESULUI LA O BAZĂ DE DATE ÎN REŢELE NOVELL, UNIX ŞI WINDOWS

C1. Implementarea primei soluţii bazate pe semafoare în reţele Novell.......................... anexaC-1
C2. Implementarea celei de-a doua soluţii bazate pe semafoare în reţele Novell................ anexaC-3
C3. Implementarea celei de-a treia soluţii bazate pe semafoare în reţele Novell................ anexaC-5
C4. Implementarea soluţiei bazate pe un proces coordonator în reţele Novell.................. anexaC-6
C5. Implementarea celei de-a treia soluţii bazate pe semafoare în reţele UNIX, şi Windows.. anexaC-13

ANEXA D. IMPLEMENTAREA SISTEMULUI DOSTP ÎN REŢELE UNIX ŞI WINDOWS

D1. Definirea clasei CObject... anexa D-1
D2. Definirea clasei CVector... anexa D-1
D3. Definirea clasei CObjectManager.. anexa D-2
D4. Definirea clasei CDispatcher.. anexa D-3
D5. Definirea macroinstrucţiunilor pentru conectarea automată a obiectelor in sistemul DOSTP.. anexa D-4

ANEXA F. COMPARAŢIE ÎNTRE PERFORMANŢELE VARIANTELOR DISTRIBUITE ŞI SEMIDISTRIBUITE ALE SISTEMULUI TGC anexa F-1

ANEXA E. COMPARAŢIE ÎNTRE PERFORMANŢELE TRANZACŢIILOR DISTRIBUITE ŞI CELE ALE TRANZACŢIILOR LOCALE ÎN SISTEMUL DOSTP anexa E-1

BIBLIOGRAFIE.. biblio-1

BUPT

INTRODUCERE

Se poate considera că aplicaţiile distribuite îşi datorează apariţia şi dezvoltarea lor actuală, progreselor notabile realizate în tehnologiile microelectronicii, calculatoarelor şi comunicaţiilor, progrese care au condus rapid la un raport performanţă/cost avantajos.

Problemele unui sistem de operare distribuit sunt parţial similare cu cele ale unui sistem de operare clasic, centralizat. Diferenţele de bază se datorează însă unor aspecte şi probleme noi apărute, diferite de cele specifice sistemelor centralizate; acestea constau în acuitatea problemelor de ***sincronizare***, care, în contextul unui sistem distribuit nu se pot rezolva fără ***comunicare***, întîrzierilor aleatorii în comunicaţii ceea ce generează, în consecinţă, natura probabilistică a informaţiilor de stare, materializate prin viziuni diferite de la proces la proces, şi nu în ultimul rînd eterogenităţii elementelor de prelucrare care compun sistemul.

Se poate aprecia că modificările în structura şi funcţiile sistemelor de operare distribuite au fost rezultatul presiunii din mai multe direcţii:

(A) pe de o parte, modificările de arhitectură ale procesoarelor au avut implicaţii majore în proiectarea sistemelor de operare. În acest sens se pot distinge, cel puţin trei aspecte majore ale modificărilor: (1) creşterea vitezei de lucru a procesoarelor, care acutizeaza problemele de acces la memorie şi timpul irosit pentru intrarea/ieşirea din nucleul sistemului de operare; (2) prelucrarea multiprocesor; (3) spaţii de adresa (virtuale) foarte mari. Un impact major l-a avut aspectul (1); acesta a generat doua tendinţe: folosirea masivă a *thread*-urilor şi deplasarea unor funcţii ale sistemelor de operare de la nivelul nucleului la nivelul utilizator; ultima tendintă furnizează sistemului de operare şi o flexibilitate crescută. Aspectul (3) implica posibilitatea folosirii unui singur spaţiu global de adrese virtuale, partajat de toate aplicatiile; aceasta implică, evident, probleme referitoare la fragmentarea spaţiului de adrese şi protecţia sa dar simplifică problemele de adresare.

(B) pe de altă parte, la cealaltă extremă, modificările calitative de la nivelul programelor de aplicaţie impun noi cerinţe sistemelor de operare. Printre acestea pot fi enumerate: (1) ***distribuţia masivă***, care ridică probleme legate de localizarea resurselor, securitate, autentificare, criptare şi eterogenitate şi ***comunicaţii de grup*** (2) ***tranzacţiile atomice***, care sunt realizate din ce în ce mai mult la nivelul sistemelor de operare distribuite, fie implementate la nivelul nucleului, fie în servere separate şi accesibile prin intermediul unor biblioteci puse la dispoziţia utilizatorilor (3) ***aplicaţiile multi-media*** care introduc o caracteristică timp-real şi deci impun reconsiderarea aspectelor specifice prelucrarii timp-real.

Lucrarea de faţă are în vedere două scopuri:

(1) un studiu teoretic asupra problematicii specifice sistemelor de operare distribuite şi interacţiunii sistem de operare distribuit -prelucrare distribuită. Am considerat necesar acest lucru, deoarece se apreciază că acest domeniu se află în prima fază a maturităţii sale, aceea de identificare a paradigmelor şi problemelor sale fundamentale fără ca acestea sa fie complet soluţionate, nici sub aspect teoretic nici sub aspect practic. Problemele clasice ale sistemelor de operare capătă o altă natură în contextul elementelor multiple de prelucrare şi al distribuirii datelor şi controlului şi, în consecinţă, necesită metode noi de soluţionare teoretică şi practică.

(2) proiectarea şi implementarea unor mecanisme suport pentru dezvoltarea de aplicaţii distribuite în reţele locale UNIX, Microsoft Windows (Win 32) şi Novell NetWare. Aceste mecanisme sunt puse la dispoziţia utilizatorilor sub forma unor biblioteci de programe care furnizeaza API-urile necesare pentru: (a) controlul accesului concurent la o bază de date (b) comunicaţii de grup (c) utilizarea unui sistem distribuit de obiecte prin intermediul tranzacţiilor atomice.

Lucrarea este structurată în două mari părţi: prima parte cuprinde aspectele teoretice referitoare la problemele specifice unui sistem de operare distribuit şi proiectarea mecanismelor suport (a) şi (b) de mai sus, iar partea a doua descrie suportul teoretic necesar proiectarii unui mecanism de tipul (c) şi apoi prezentarea soluţiilor de proiectare şi implementare a acestuia în reţele locale.

Cuprinsul celor unsprezece capitole, în care este structurată lucrarea, este următorul:

BUPT

Partea I: SISTEME DISTRIBUITE

Capitolul 1: Sisteme de operare distribuite. Suportul oferit prelucrarilor distribuite: probleme specifice, modele de proiectare.

In primul capitol se urmăreste stabilirea unei definiţii a sistemelor de operare distribuite, se precizează obiectivele generale urmarite de sistemele cu prelucrare distribuită şi criteriile pe baza cărora se definesc sistemele distribuite şi se reliefează principalele avantaje ale sistemelor distribuite în raport cu sistemele centralizate. Sunt prezentate de asemenea, modelele actuale din prelucrarea distribuită, modelul ierarhic, modelul federativ, modelul partajării resurselor, modelul omogen şi modelul fluxului de date, termenul de prelucrare distribuită fiind asimilat în accepţiunea de *execuţie a unui program, sub forma unor procese concurente, în mai multe noduri*. Este prezentat de asemenea, comparativ, specificul sistemelor de operare pentru reţelele de calculatoare şi cel al sistemelor de operare distribuite. Sunt analizate funcţiile de gestiune a proceselor şi de gestiune a fişierelor ale sistemelor de operare distribuite şi sunt prezentate cele două modele de proiectare ale unui sistem distribuit: ***modelul proces*** şi ***modelul obiect***, diferenţele majore între cele două modele de proiectare constînd în modul în care sunt implementate conceptul de entitate funcţională şi sincronizarea. O atenţe deosebită este acordată interacţiunii dintre sistemul de operare distribuit - prelucrare distribuită. In acest sens, sunt prezentate două exemple de suporturi software puse la dispoziţia utilizatorilor pentru dezvoltarea de aplicaţii distribuite. Un prim exemplu îl constituie mediul OSF DCE; sunt arătate arhitectura generala DCE, serviciile fundamentale DCE şi interacţiunea dintre servicii. Cel de-al doilea exemplu este sistemul TDL, un suport software proiectat pentru conectarea programelor în reţele UNIX, cunoscîndu-se modul de comunicare al acestora în cadrul unei aplicaţii.

Capitolul 2. Comunicaţii în sisteme distribuite. Protocoale.

Obiectivul capitolului 2 îl constituie problemele comunicaţiei în reţele de calculatoare, ale implementării software-ului de comunicaţie precum şi aspecte specifice ale protocoalelor de comunicaţie în sisteme distribuite. Este prezentat modelul arhitectural ISO/OSI pentru interconectarea sistemelor deschise, insistîndu-se pe importanţa pe care o prezintă nivelul transport pentru sistemele de operare, fiind furnizorul principal de servicii. Se arată două soluţii de implementare a modelului OSI şi importanţa modelului OSI pentru sistemele distribuite. Ca exemple de protocoale utilizate în reţelele locale de calculatoare şi folosite şi în suporturile software proiectate şi prezentate de autor pe parcursul lucrării, sunt amintite pe scurt protocolul IP, insistîndu-se pe propunerea IP versiunea 6, TCP, IPX/SPX realizîndu-se legătura cu modul de utilizare al acestora în sistemele de operare UNIX, Windows şi Novell NetWare. Comparativ cu protocoalele cu caracter univeral folosite în reţelele de calculatoare sunt reliefate şi aspectele specifice ale suportului de comunicaţie pentru sistemele distribuite.

Capitolul 3. Paradigme ale interacţiunii dintre procese în sisteme distribuite.

Capitolul 3 este dedicat prezentării paradigmelor interacţiunii dintre procese în sisteme distribuite. Este prezentat ***modelul client-server*** insistîndu-se pe proiectarea serverelor; se arată situaţiile în care trebuie folosit un server concurent multiproces, un server concurent cu ***select***, sau un server iterativ, facîndu-se o comparaţie între performanţele acestora. Modelul client-server poate fi implementat utilizînd ambele clase de protocoale prezentate în capitolul 2: protocoale cu caracter universal şi protocoale cu caracter restrîns. Pentru obţinerea de performanţe superioare este necesar însă, un protocol specific, deci cu caracter restrîns; se exemplifică schimbul de mesaje necesar intr-un astfel de protocol client-server.

Un alt aspect descris în acest capitol este cel al comunicării prin mesaje între procese; sunt prezentate problemele specifice ale comunicării prin mesaje, primitive cu şi fără blocare, primitive cu şi fără blocare, sigure şi nesigure, forme structurate ale comunicarii prin mesaje şi un exemplu de implementare a comunicaţiei prin porturi - comunicaţia din sistemul Mach.

Ca o altă paradigmă a interacţiunii dintre procese în sisteme distribuite este prezentată comunicarea la distanţă între procese prin conducte, exemplificările fiind facute în Microsoft Win32 în sistemul de operare Windows NT şi Windows 95, în ultimul nefiind însă implementată decît partea de client; totodată se propune o soluţie pentru utilizarea conductelor în servere multifir în Win32.

Modelul apelului de procedura la distanţă este dezbătut pe larg, avîndu-se în vedere aspecte precum transmiterea parametrilor şi rezultatelor, legarea clientilor la servere, probleme de concurenţă, semantica RPC în prezenţa defectelor, protocoale pentru RPC. Sunt prezentate şi extensii ale mecanismului de bază RPC şi analizate aspectele critice în proiectarea comunicaţiei prin RPC. Exemplificările sunt facute pentru SUN RPC, DCE RPC şi Microsoft RPC.

BUPT

Ca un ultim model de comunicaţie în sistemele distribuite sunt prezentate comunicaţiile de grup, cu exemplificări pentru MCL, ISIS şi Amoeba.

Capitolul 4. Sincronizarea în sisteme de operare distribuite.

Capitolul 4 analizează una dintre cele mai importante probleme în sistemele distribuite: aceea a sincronizării proceselor. Termenul de ***sincronizare*** este folosit în sisteme distribuite pentru a referi trei probleme distincte, dar în ultimă instanţă înrudite: (1) sincronizarea intre emiţătorul şi receptorul unui mesaj; (2) specificarea şi controlul acţiunilor unor procese care cooperează în cadrul unei aplicaţii distribuite; (3) serializarea accesului concurent la obiectele partajate între mai multe procese, problemă cunoscută şi în bazele de date sub numele de ***controlul concurenţei***. In capitolul 4 se trateaza aspecte privitoare numai la problemele de tipul (2); problemele de tipul (1) au fost dezbatute în capitolele 2 şi 3 iar problemele de tipul (3) vor fi abordate în capitolul 7.

Mecanismele de sincronizare pentru sistemele distribuite vor trebui să determine, printr-o metoda oarecare, o ordonare totala sau parţiala a unui set de evenimente. Capitolul 4 prezintă mai întîi aspecte ale ordonarii evenimentelor într-un sistem distribuit şi apoi, pe baza acestora, două clase mari de mecanisme de sincronizare: ***centralizate*** şi ***distribuite***. Unele dintre aceste mecanisme se bazează pe conceptul de ***excludere reciproca*** care presupune o ordonare a unor secvenţe de cod executate de procesele concurente; aceste mecanisme pot utiliza diferite tehnici precum: un ***proces central coordonator*** sau un ***token circulant***. Alte mecanisme se bazează pe înregistrarea evenimentelor semnificative în cursul unor prelucrări asincrone; acestea utilizează obiecte precum ***contori de evenimente*** şi ***secvenţiatori*** sau ***semafoare distribuite***. Sistematizarea algoritmilor distribuiţi pentru sincronizare s-a facut în capitolul 4 astfel: (a) în clasa algoritmilor distribuiţi pentru excludere reciprocă s-au prezentat algoritmul lui Lamport, algoritmul lui Ricard şi Agrawala şi algoritmul Raymond, pentru excludere reciprocă în accesul la K resurse identice (b) în clasa algoritmilor care utilizează un token circulant a fost prezentat un algoritm pentru sisteme în care nodurile sunt ordonate într-un inel logic şi algoritmul Chandy (pentru sisteme în care nodurile nu sunt ordonate într-un inel logic) precum şi o variantă de implementare a conceptului de secvenţiator circulant; (c) în clasa algoritmilor care utilizează transmisii broadcast s-au proiectat şi implementat semafoare distribuite, contori de evenimente distribuiţi şi secvenţiatori distribuiţi.

Se remarcă faptul că, pentru toţi algoritmii prezentaţi, au fost realizate implementări în reţele locale Unix, Microsoft Windows şi Novell NetWare. Unele dintre aceste implementări (din lipsă de spaţiu, nu toate implementările) sunt prezentate în anexe. Testele s-au facut în retele mixte Unix + Windows, respectiv reţele Novell.

În final, în capitolul 4 se prezintă doi algoritmi de alegere a coordonatorului: algoritmul Bully şi algoritmul în inel.

Capitolul 5. Proiectarea şi implementarea unor mecanisme suport pentru dezvoltarea de aplicaţii în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare.

În finalul primei părţi a lucrarii, în capitolul 5 se prezintă proiectarea şi implementarea a două mecanisme suport pentru dezvoltarea de aplicaţii în reţele UNIX, Microsoft Windows (Win32) şi Novell NetWare: primul mecanism are ca scop controlul accesului concurent la o bază de date iar cel de-al doilea constă într-un toolkit pentru comunicaţii de grup. Proiectarea şi implementarea acestor mecanisme foloseşte conceptele teoretice prezentate în capitolele anterioare: protocoale de comunicaţii prezentate în capitolul 2 (TCP, UDP, IPX), comunicare prin mesaje, concurenţa la nivelul serverelor, comunicaţii de grup (capitolul 3), algoritmi pentru obtinerea excluderii reciproce, semafoare distribuite (capitolul 4).

Pentru controlul accesului concurent la o baza de date s-au propus patru soluţii: trei soluţii bazate pe semafoare şi o soluţie bazată pe un proces coordonator. Toate soluţiile au fost implementate în reţele mixte Unix + Windows (Win32) şi Novell NetWare. Pentru soluţiile bazate pe semafoare în reţele Unix şi Windows s-au folosit semafoarele distribuite, proiectate şi implementate în capitolul 4, iar pentru reţele Novell NetWare s-a folosit faptul ca sistemul de de operare NetWare pune la dispoziţia programelor de aplicaţie conceptul de semafor. Pentru proiectarea unui toolkit pentru comunicaţii de grup s-au propus patru variante: o variantă semidistribuită, două variante complet distribuite în care în una se foloseşte transmisia broadcast şi o variantă în care nodurile sunt organizate într-un inel logic. Se prezintă în detaliu sintaxa API-urilor puse la dispoziţia programelor utilizator, în cadrul cerinţelor impuse proiectării şi se face o comparaţie a celor patru variante.

BUPT

Partea a II-a: CRITERII DE PROIECTARE A SISTEMELOR DE PROGRAMARE DISTRIBUITĂ BAZATĂ PE OBIECTE (SPDBO). PROIECTAREA UNUI SUPORT SOFTWARE BAZAT PE MODELUL TRANZACŢIILOR ATOMICE PENTRU SPDBO ÎN REŢELE UNIX ŞI WINDOWS.

Capitolul 6. Sisteme de programare distribuită orientată pe obiecte (SPDBO). Criterii de proiectare

Partea a II-a a lucrarii cuprinde suportul teoretic necesar pentru proiectarea sistemelor de programare distribuită bazată pe obiecte precum şi proiectarea şi implementarea în reţele locale a unui suport software pentru lucrul cu tranzacţii atomice în aplicaţii distribuite.

Partea a II-a debutează cu capitolul 6 în care se prezintă problemele majore care trebuie avute în vedere pentru proiectarea unui sistem de programare distribuită. Sunt reliefate aspecte ale componenţei şi structurii obiectelor, ale gestionarii obiectelor incluzînd probleme de sincronizare ale operaţiilor asupra obiectelor şi probleme de siguranţă a obiectelor. Un subcapitol special este dedicat interacţiunii obiectelor, cuprinzînd probleme de localizare a obiectelor şi al modului de apel al operaţiilor. Un alt subcapitol este destinat modului de reprezentare a obiectelor în memorie; un accent mai mare este pus pe reprezentarea obiectelor în memoria secundară, deoarece acest lucru este legat de recuperabilitatea obiectelor, o cerinţă esenţtală într-un sistem de programare distribuită bazat pe obiecte.

Capitolul 7. Controlul concurenţei şi recuperarea obiectelor

Se apreciază că proiectarea aplicaţiilor distribuite care să ofere siguranţă în funcţionare poate fi obţinută prin două metode generale, conceptual diferite: (1) prima metodă utilizează conceptul de **tranzacţie atomică**; (2) a doua metodă utilizează copii multiple ale obiectelor necesare aplicaţiilor.

Deoarece în această lucrare se are în vedere prima metodă, în capitolul 7 se prezintă întîi modelul de lucru bazat pe tranzacţii, apoi soluţii pentru realizarea atomicităţii tranzacţiilor în sistemele distribuite; în acest sens este prezentat protocolul ***two-phase-commit*** şi modul lui de implementare în OSI (Commitment Concurrency and Recovery).

Se prezintă apoi tehnici pentru recuperarea obiectelor, precum ***Value-Logging*** sau ***Operation Logging, Redo-Only Logging*** sau ***Undo-Only Logging*** şi structura generală a subsistemului de recuperare.

Un subcapitol important din capitolul 7 este dedicat controlului concurenţei. După prezentarea cerinţelor mecanismelor de control al concurenţei şi al conceptului de serializare al tranzacţiilor se analizează mecanismele tradiţionale pentru controlul concurenţei: mecanismele bazate pe blocarea cu ***lock***-uri, mecanismele bazate pe marca de timp, şi variante de control optimist al concurenţei. Se discută apoi extinderea mecanismelor tradiţionale pentru controlul concurenţei - mecanismul ***blocării altruiste*** şi ***mecanismul validarii instantanee*** - şi se analizează mecanismele pentru controlul concurenţei bazate pe informaţii semantice.

Capitolul 8. Controlul interblocarii obiectelor

Capitolul 8 trateaza controlul interblocarii în sistemele distribuite unde, datorită dispersiei informaţiilor pe mai multe noduri, interblocările sunt mai greu de prevenit, evitat şi detectat. Se prezintă modelul general utilizat pentru tratarea interblocării, metodele de prevenire ale interblocărilor, care se bazează pe tranzacţii atomice şi pe ordonarea tranzacţiilor prin asocierea de mărci de timp şi metodele de detecţie a interblocării. Algoritmii de detecţie se clasifică în trei mari grupuri: algoritmi centralizaţi, algoritmi ierarhici şi algoritmi distribuiţi; ultimii, la rîndul lor se subîmpart în următoarele clase de algoritmi: algoritmi care transmit caile grafului dependenţelor tranzacţiei, algoritmi care urmăresc arcele, algoritmi cu difuzie şi algoritmi care stabilesc starea globală.

Pentru fiecare grup de algoritmi se prezintă cîte un exemplu; pentru algoritmii complet distribuiţi se prezintă două exemple: un algoritm **edge-chasing** pentru detecţia interblocării în modelul AND şi un algoritm cu difuzie pentru detectia interblocării în modelul OR.

Capitolul 9. Proiectarea unui suport software pentru programare distribuită bazată pe obiecte şi pe trazacţii atomice (DOSTP)

Capitolul 9 cuprinde descrierea elementelor componente, a funcţionării şi a modului de proiectare al unui suport software pentru un sistem de programare distribuită. Sunt prezentate în primul subcapitol cerinţele impuse: suportul are drept scop principal furnizarea unor primitive pentru lucrul cu tranzacţii atomice asupra unor obiecte dispersate în reţea, în condiţiile asigurării recuperabilităţii obiectelor. Se arată (în contextul conceptelor teoretice prezentate în capitolele 6,7 şi 8) că sistemul proiectat (DOSTP) utilizează ca model de arhitectură *modelul integrat* şi varianta *dinamică a modelului obiect activ*. Schema de sincronizare a operaţiilor la nivelul unui obiect

BUPT

este o schemă optimistă, iar pentru recuperarea obiectelor se foloseşte protocolul *write-ahead logging* şi tehnicile *Value-logging* şi *Redo-only-logging*. Pentru interzicerea restartării tranzacţiilor la infinit, s-a optat pentru implementarea unui mecanism *Wound-die*. După prezentarea arhitecturii generale şi a modelului pentru sistemul DOSTP sunt descrise apoi componentele sistemului DOSTP: serverele de comunicaţii, managerii de tranzacţii, managerii de obiecte, serverele de nume, managerul central, procesele utilizator care realizează tranzacţiile. Sistemul de recuperare al obiectelor nu este implementat în DOSTP ca proces distinct; el se compune din module plasate în managerii de obiecte şi în managerii de tranzacţii şi din fişierele lor *log* aferente.

Capitolul 10. Implementarea sistemului DOSTP în reţele Unix şi Microsoft Windows (Win32)

Implementarea sistemului DOSTP s-a făcut, atît în varianta Unix cît şi în varianta Windows urmărind îndeaproape proiectarea prezentată in capitolul 9. Există totuşi diferenţe între cele două variante, generate în special de faptul că în varianta Unix elementele componente sistemului au fost realizate ca procese distincte iar în Windows au fost realizate ca *thread*-uri. Capitolul 10 are ca centru de greutate proiectarea în manieră obiectuală a managerilor de obiecte. Scopul principal în realizarea managerilor de obiecte este ascunderea detaliilor de conectare cu sistemul tranzacţional. Managerii de obiecte sunt realizaţi astfel încît ei comunică numai cu serverul de comunicaţie local, fară a avea cunostinţă dacă serviciile le-au fost solicitate local sau de la distanţă. In capitolul 10 sunt descrise, pentru un caz particular de obiect, mecanismele de control al concurenţei şi de recuperare ale obiectelor de acest tip, şi, pe de altă parte, se arată cum trebuie să fie structurată o aplicaţie care să folosească într-o tranzacţie obiecte de acest tip. Un alt aspect important, prezentat în detaliu în capitolul 10, este circulaţia mesajelor în sistem, în cele şase situaţii distincte care pot să apară în funcţionarea sistemului DOSTP: (1) conectarea unui nod la sistemul DOSTP; (2) integrarea sau eliminarea unui obiect din sistem; (3) iniţierea unei noi tranzacţii; (4) implicarea unui obiect local şi a unui obiect de la distanţă într-o tranzacţie; (5) realizarea protocolului *two_phase commit*; (6) recuperare după o defecţiune a unui nod şi a reintrării sale în sistem.

Capitolul 11. Concluzii şi contribuţii originale.

Capitolul 11 al lucrării prezintă contribuţiile originale pe care le aduce autorul în cadrul tezei şi unele elemente referitoare la valoarea aplicativă şi a direcţiilor de dezvoltare viitoare a cercetărilor efectuate.

Lucrarea conţine patru anexe, in care sunt descrise aspecte de detaliu din capitolele 4,5, 9 si 10.

BUPT

BUPT

Capitolul 1.

SISTEME DE OPERARE DISTRIBUITE. SUPORTUL OFERIT PRELUCRĂRILOR DISTRIBUITE, PROBLEME SPECIFICE, MODELE DE PROIECTARE

În acest capitol sunt prezentate aspecte legate de specificul sistemelor de operare distribuite în raport cu cele pentru reţelele de calculatoare, aspecte legate de interacţiunea sistem de operare distribuit - aplicaţie distribuită, probleme relative la gestiunea proceselor şi fişierelor într-un sistem distribuit (problemele legate de comunicare şi sincronizare într-un sistem distribuit sunt prezentate mai pe larg în capitolele urmatoare) iar în final se prezintă modelele uzuale de proiectare a unui sistem distribuit: *modelul proces* şi *modelul obiect.*

1.1. Introducere

Un *sistem de operare* poate fi definit ca o colecţie organizată de programe care indeplinesc două mari funcţii:
1. gestionează resursele hardware ale sistemelor de calcul, implementînd algoritmi prin care se încearcă o optimizare a performanţelor; resursele gestionate sunt: procesoarele, memoria, dispozitivele de intrare/ieşire, fişierele, liniile de comunicaţie.
2. crează o interfaţă între utilizator şi sistemul de calcul, extinzînd setul de operaţii disponibile utilizatorului, minimizînd efortul uman de programare şi simplificînd modul de lucru cu calculatorul prin automatizarea operaţiilor uzuale în exploatarea echipamentelor.

Aceste funcţii presupun alocarea şi partajarea resurselor hardware între utilizatori, prevenirea interferenţei între utilizatori, facilitarea executării operaţiilor de intrare/ieşire, recuperarea erorilor, contabilizarea utilizării resurselor sistemului de calcul, facilitarea execuţiei de operaţii concurente în sistem, organizarea datelor într-o manieră optimă în vederea garantării securitaţii lor şi accelerării accesului la ele, gestiunea liniilor de comunicaţie în reţele de calculatoare, implementarea unui limbaj de comandă prin care sistemul interacţionează cu utilizatorii.

Evoluţia de peste 40 de ani a sistemelor de calcul s-a reflectat şi în sistemele de operare în aşa fel încît se poate spune că sistemele de operare au urmărit îndeaproape dezvoltarea echipamentelor de calcul. Reţelele locale de date devenit, în urma evoluţiei echipamentelor de calcul, cea mai avansată transpunere în practică, pe sisteme operaţionale, a conceptelor prelucrării complet distribuite. În utilizarea reţelelor proliferează aplicaţii precum transferul fişierelor, poşta electronică, utilizări ale bazelor de date. Lucrul în reţea impune un nou model în proiectarea sistemelor de operare: *sistemul client-servant.* Clienţii, distribuiţi oriunde într-o reţea de calculatoare, utilizează servicii puse la dispoziţie de servanţi (componente software/hardware ale reţelei); servanţii sunt în general dedicaţi unui anumit tip de serviciu specific, precum tipărire, accesul la fişiere, la baze de date, la informaţiile de adresare în sistem. Sistemele de operare pentru sistemele distribuite au ca obiectiv major asigurarea unei interfeţe unitare şi omogene pentru utilizator, prin intermediul căreia acesta să poată avea acces la totalitatea resurselor sistemului, independent de gradul de eterogenitate al acestuia. Astfel, sistemele de operare încorporează totalitatea mecanismelor necesare pentru execuţia distribuită a unei aplicaţii scrise într-un limbaj de programare de nivel înalt.

Se propune urmatoarea definiţie pentru un *sistem de operare distribuit* ([Mus93d][DPV88]):

Definiţie:

Se numeşte *sistem de operare distribuit,* un sistem de operare care este executat pe mai multe unităţi de prelucrare, independente, şi care trebuie să asigure următoarele:
(1) controlul alocării resurselor mediului distribuit, pentru a permite utilizarea lor într-un mod cît mai eficient; (2) perceperea mediului distribuit de către utilizatori ca pe un procesor virtual şi nu ca pe o colecţie de unităţi de prelucrare distincte, conectate printr-un sistem de comunicaţie; acest procesor virtual va servi ca suport utilizatorilor pentru dezvoltarea programelor în limbaje de nivel înalt;
(3) comunicaţii sigure între unităţile de prelucrare din cadrul mediului distribuit;
(4) ascunderea dispersării resurselor;
(5) furnizarea unor mecanisme de protecţie a resurselor impotriva acceselor neautorizate.

BUPT

1.2. Prelucrarea distribuita a datelor

Anii 1990, cel puțin pînă în momentul de față, au marcat începutul perioadei calculului distribuit. Prelucrările se execută în paralel fie pe sisteme multiprocesor, fie în rețele de calculatoare. Rețelele de calculatoare pot fi configurate dinamic, pot fi interconectate. Interconectivitatea este facilitată prin introducerea conceptului de *sistem deschis*.

Un *sistem distribuit* este format dintr-un ansamblu de noduri legate între ele prin linii de comunicație; acestea reprezintă unica modalitate de comunicație între noduri. Fiecare nod este compus din unul sau mai multe procesoare cu acces la o memorie comună.

Sistemele distribuite sunt realizate ca un ansamblu unitar hardware-software care prezintă atît o *distribuire fizică cît și logica*. Enslow [DPV88] sugerează un spațiu tridimensional de distribuire pentru această clasă de sisteme: distribuirea la nivelul hardware-ului, distribuirea controlului și distribuirea bazelor de date. Conform definiției adoptată de el, un sistem cu prelucrare distribuită a datelor trebuie să posede cinci proprietăți:

1. *Dispersarea geografică* a resurselor fizice și logice aflate în legatură permanentă.

2. *Descentralizarea controlului și a informației de stare*; este o proprietate care conduce la deosebiri importante între sistemele de operare ale sistemelor centralizate și cele distribuite. Abordările de pînă acum în ceea ce privește sistemele de operare pentru rețelele locale de calculatoare au urmat două mari direcții: preluarea unui sistem de operare multiprogramat, tradițional și deci cunoscut, căruia i se adaugă capacități de comunicare în rețea sau dezvoltarea, de la zero, a unui nou sistem de operare.

3. *Multitudinea de resurse de prelucrare fizice și logice* utilizabile concomitent și care pot fi alocate dinamic unor sarcini specifice. Aceste resurse pot fi procesoare de diverse tipuri și fabricații conectate între ele, în diverse configurații și arhitecturi; de asemenea se pot utiliza (static sau dinamic) echipamente periferice diverse, partajate sau alocate corespunzător. Corespunzător multitudinii de resurse fizice, există în sistemele distribuite o varietate de resurse logice (software de comunicație, sisteme de gestiune a fișierelor sau bazelor de date, software suport pentru dezvoltarea de programe) care conlucrează la rezolvarea sarcinilor specifice ale aplicațiilor în context distribuit.

4. *Autonomia cooperativa a procesoarelor și a resurselor logice*, care se exprimă prin posibilitatea unei resurse de a rezolva in mod independent o cerere de serviciu locală și de a rezolva prin cooperare cu celelalte resurse o cerere de serviciu globală. Autonomia procesoarelor este o caracteristică fundamentală care decide, în ultimă instanță apartenența unui sistem la clasa sistemelor distribuite. Componentele de prelucrare trebuie să fie autonome. Ele cooperează cu alte componente, dar nu le controlează. Eventualele situații de concurență se rezolvă prin algoritmi specializați care se execută de către fiecare componentă de prelucrare. Necesitatea sincronizării activităților în sistemul distribuit, pentru evitarea conflictelor, constituie o altă diferență importantă față de sistemele clasice cu control centralizat.

5. *Transparența la nivelul utilizatorilor* care cer servicii prin nume, fără a se preocupa de resursele care participă la realizarea serviciului cerut. Software-ul sistemelor distribuite identifică fizic o resursă necesară unui anumit serviciu (acțiune) indiferent de locația geografică a acesteia.

1.2.1. Obiectivele sistemelor distribuite

Apariția și dezvoltarea sistemelor distribuite a răspuns unor necesități ivite în domeniul prelucrării automate a datelor. Avîntul din ultimii ani a unor astfel de sisteme a fost posibil datorită progreselor rapide înregistrate în domeniul echipamentelor de calcul și a celor de telecomunicație. Avantajele principale ale sistemelor distribuite in raport cu sistemele centralizate sunt următoarele:

1. *Autonomia locală combinată cu posibilitatea de comunicare.* O caracteristică generală în exploatarea sistemelor de calcul s-a constatat a fi [BW86] tendința utilizatorilor de a ezita in incredințarea informațiilor proprii sistemelor mari, centralizate, care deseori, pot compromite securitatea datelor; pe de altă parte, aplicații de largă utilizare, ca de exemplu cele financiare, este de dorit să se desfășoare pe grupuri de utilizatori. O altă problemă dificil de soluționat în folosirea unui sistem centralizat s-a dovedit a fi impartirea costurilor legate de acesta. Autonomia locală a sistemelor distribuite elimină aceste impedimente si, pe de altă parte, cu ajutorul facilităților de comunicație se realizează transmiterea de informații la distanță. Un alt factor subiectiv, care a impulsionat descentralizarea prelucrării, il constituie necesitatea punerii în valoare a unor caracteristici umane specifice activității creatoare: capacitatea de acțiune independentă, sentimentul competiției in dialogul direct cu calculatorul. Acestea sunt puse mai bine in valoare în grupuri relativ mici decit in cele foarte mari, impuse de caracterul tehnico-organizatoric complex al marilor sisteme centralizate.

2. *Utilizarea în comun a unor resurse.* Aceste resurse pot fi procesoare cu anumite caracteristici specifice, echipamente periferice, fișiere și baze de date. Un proces poate migra din procesorul local pe un alt nod dacă nodul local nu poseda resursele cerute, dacă prezintă defectuni în funcționare, sau dacă pe un nod situat la distanță se obține un timp de răspuns mai bun. Gestiunea ansamblului de resurse utilizate in comun se face in scopul optimizării folosirii lor, optimizării timpului de răspuns, controlului incărcării rețelei.

3. *Siguranta in funcționare.* Existența unui numar mare de procesoare face posibilă continuarea activității unui sistem distribuit și in cazul in care un procesor își incetează funcționarea corectă. Această facilitate este prezentă și la nivelul software: in acest scop, fișierele și bazele de date pot fi replicate, pachetele de programe software pot fi accesibile in mai multe noduri, calcule executate parțial pot fi continuate pe alt nod.

BUPT

4. *Performante superioare.* Conexiuni multiple între diversele procesoare permit execuţia concurentă a unor porţiuni din programe (procese) ceea ce, evident, va permite scurtarea duratei de execuţie totale a programului. Localizarea datelor şi procesoarelor disponibile se face după principiul *minimei apropieri* faţă de nodul local.

5. *Extensibilitate şi adaptabilitate a configuraţiilor* (atît hardware cit şi software). Un sistem distribuit poate fi mult mai uşor modificat decît un sistem centralizat, prin adăugarea sau îndepărtarea unor componente, datorită modularităţii sale, atît în ceea ce priveşte structura hardware cit şi cea software. Modificările vizează marirea performanţelor sistemului.

În aceste condiţii, este motivată tendinţa actuală ca sistemele distribuite să înlocuiască, treptat sistemele centralizate. Prelucrarea distribuită îşi propune să pună în valoare în modul cel mai eficient, avantajele descentralizării prelucrării, astfel încît să răspundă necesităţilor utilizatorului şi aplicaţiilor sale, fară pierderea controlului asupra standardelor, calităţii datelor, procedurilor de colectare, manipulare şi prelucrare a unor volume mari de date, avantaje specifice sistemelor centralizate. În acelaşi timp, prelucrarea distribuită reprezintă un mijloc de a satisface dorinţa utilizatorului de apropiere faţă de sistemul de calcul, evitînd dezavantajele unei descentralizări excesive.

In formularea criteriilor pe baza cărora se definesc actual sistemele cu prelucrare distribuită s-au avut în vedere de la început o serie de obiective pe care acestea trebuie să le atinga. Acestea sunt:
(1) siguranţă în funcţionare sporita;
(2) disponibilitate mărită în cazul defactării unor componente;
(3) grad înalt de modularitate;
(4)-perfomanţe de sistem mai ridicate;
(5) timp de răspuns bun în cazul supraîncărcărilor temporare;
(6) utilizarea în comun a resurselor.

În prezent, pentru ca toate aceste facilităţi să poată fi disponibile, proiectantii de sisteme distribuite sunt confruntaţi cu probleme vizînd următoarele direcţii: (1) comunicaţia dintre procesoare şi sincronizarea acestora; (2)gestiunea resurselor din sistem; (3) distribuirea resurselor; (4) distribuirea încarcării sistemului, care include rezolvarea problemei planificării globale a proceselor pe nodurile sistemului; (5) gestiunea proceselor; (6) gestiunea operaţiilor de intrare-ieşire; (7) gestiunea memoriei; (8) controlul sistemului distribuit astfel încît să se conserve, caracteristica sa cea mai importantă: fiabilitate ridicată şi disponibilitate în cazul defectării unor componente; (9)securitatea informaţiilor; (10) mascarea eterogenităţii sistemului (aparută datorită eterogenităţii procesoarelor din nodurile reţelei si diferitelor convenţii de reprezentare a datelor); (11) existenţa unor limbaje de programare care să includă posibilitatea de distribuire a proceselor şi realizarea unei autonomii cooperative transparente la nivelul utilizatorului.

1.2.2. Modele actuale în prelucrarea distribuită

Termenul de *prelucrare distribuită* semnifică execuţia unui program sub forma unor procese concurente, în mai multe noduri.

În prezent, în prelucrările distribuite sunt folosite următoarele modele: *modelul ierarhic, modelul federativ, modelul partajării resurselor, modelul omogen şi modelul fluxului de date:*

1. *Modelul ierarhic* este specific sistemelor distribuite în timp real. El presupune un nivel inferior în cadrul căruia se colectează datele de intrare şi se controlează dispozitivele fizice care achiziţionează datele; nivelul imediat următor realizează gestiunea datelor şi analizează informaţiile de stare în conformitate cu anumite cerinţe, putînd iniţia chiar o reconfigurare a nivelului inferior în cazul detectării unor valori diferite de cele normale; nivelul superior este destinat controlului interactiv de către un operator uman.

2. *Modelul federativ* se compune dintr-un nivel centralizat de macroprocese şi mai multe nivele de miniprocese distribuite; poate fi considerat ca un tip special de relaţie *master-slave* . Acest model încearcă să execute în nodul local cit mai multe procese; la depaşirea capacităţii nodului, procesele aditionale sunt executate la distanţă, în alte noduri iar la terminarea prelucrării, rezultatele sunt redirectate către nodul iniţiator al prelucrării. Caracteristica principală a acestui model este coexistenţa mai multor noduri centrale fiecare maximizînd la nivelul său prelucrările de tip *master-slave* . Modelul poate plasa procese la distanţă şi la cerere. O extensie a modelului este cea în care controlul execuţiei se face centralizat, într-un nod, în timp ce procesele *slave* sunt distribuite astfel încît, prin execuţia lor concurentă, să se obţină o prelucrare rapida.

3. *Modelul partajării resurselor* pleacă de la ipoteza că toate resursele (procesoare, periferice) sunt date, în mod singular, ca disponibile pentru utilizatori care, după alocare, le utilizează conform cerinţelor lor. Adiţional, se introduc ca resurse partajabile anumite funcţii software generice, prevazute cu interfeţe standard. In acest model, utilizatorul execută prelucrările asupra resurselor printr-o secvenţă de apel a acestor funcţii. Modelul necesită un sistem de operare global care să controleze partajarea diverselor elemente distribuite din sistem şi conduce la noţiunile de obiect, gestiunea obiectelor şi programare la nivel de obiecte.

4. *Modelul omogen* se aplică unui ansamblu de calculatoare aproximativ de aceleaşi caracteristici şi resurse. In acest model se urmăreste o distributie cit mai mare a proceselor generate de un program. Distributia proceselor se face pe baza unui algoritm care minimizează (sau maximizează) diferiti parametri precum: *gradul de paralelism, încărcarea egală a procesoarelor, timpul de răspuns, încărcare maximă pe un cluster.*

5. *Modelul fluxului de date* presupune prelucrarea datelor în etape (începînd de la dispozitivele de intrare, datele sunt procesate de dispozitive specifice pînă la obţinerea rezultatului). Acest tip de prelucrare distribuită necesită limbaje specializate şi chiar procesoare specializate.

613.451
366 B

BUPT

1.2.3. Interacţiunea prelucrare distribuită - sistem de operare

O tendinţă generală actuală în folosirea reţelelor atît locale cit şi generalizate este de a se cupla staţii de lucru sau calculatoare de diverse fabricaţii, eventual cu sisteme de operare diferite. Utilizatorii unui astfel de mediu eterogen sunt confruntaţi cu probleme referitoare la alocarea resurselor la diversele procese (în special procesoare de diferite caracteristici). . Ei trebuie să creeze procese, să aloce procesoare la procese, să specifice modul de comunicaţie între procese. Aceste activităţi constituie *programarea de la nivelul comutării proceselor* (PMS=Procesor-Memory Switch Level Programming) faţă de programarea *tradiţională* plasată la nivelul programării procesorului (ISP-Instruction Set Procesor Programming). Intr-un mediu eterogen, procesoarele nu sunt singura resursă critica; în afară de procesoare specializate, mai trebuie alocate şi alte resurse precum linii de comunicaţie, bufere de date (fig 1.2.3.1).

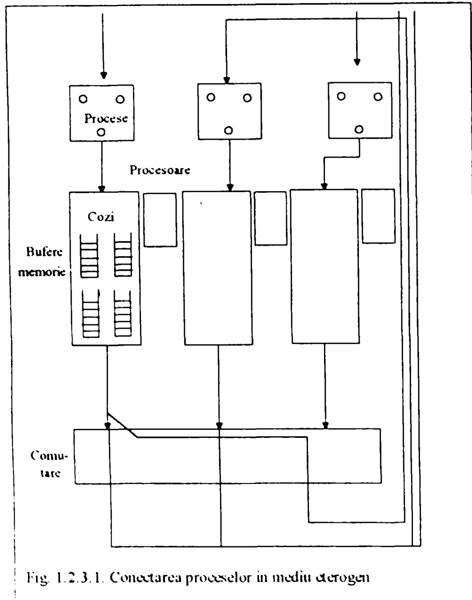


Fig. 1.2.3.1. Conectarea proceselor in mediu eterogen

În mod curent, utilizatorii unui mediu eterogen dezvoltă programe în manieră tradiţională: se scriu procese separate ca programe în diferite limbaje (Pascal,C) implementate pe anumite procesoare şi apoi printr-un program separat se încarcă aceste programe pentru execuţie. Deseori, aceste programe sunt scrise astfel încît înglobează în ele specificaţii privind dispunerea lor pe procesoare sau chiar privind structura reţelei. Acest mod de lucru face dificilă sau chiar imposibilă utilizarea aplicaţiei într-un mediu diferit, complică o dezvoltare ulterioară a aplicaţiei care ar schimba structura programelor şi suportul pentru siguranţă la defecte (prin restartarea unei noi copii a programului pe un procesor diferit). O manieră nouă de abordare a acestor probleme este de a separa activitatea de programare a programelor de ativitatea de cooperare a lor în codul aplicaţiei.Vor exista astfel două nivele de programare în cadrul unei aplicaţii: *un nivel de programare al programelor componente ale unei aplicaţii (ISP) şi un nivel de programare a aplicaţiei care utilizează programele dezvoltate anterior (PMS).* [BDW89]

Dezvoltarea suportului software pentru nivelul PMS este, calitativ, mult diferită faţă de nivelul IPS: necesită alte limbaje, metodologii, chiar medii de programare şi, pe de altă parte, interacţiunea cu sistemul de operare distribuit.

Pot fi enumerate următoarele realizari notabile în această direcţie:

1.*Limbajul Durra* este proiectat pentru a constitui suportul software al nivelului PMS [BDW89]. Limbajul oferă posibilitatea dezvoltării aplicaţiilor prin rafinari succesive după modelul metodei Spiral [BDW89]. Utilizatorii modelului Spiral identifică selectiv componentele de risc ale aplicaţiei, stabilesc cerinţele lor, şi apoi realizează fazele de proiectare, codificare şi testare. Pe parcursul desfăşurării acestui proces pot fi identificate şi alte componente de risc cărora li se poate da o prioritate mai mare. Limbajul asigură construcţii specifice pentru simularea unei aplicaţii, plecînd de la o sumară decompozitie în taskuri specificate prin interfaţa lor cu celelalte taskuri şi prin caracteristici ale execuţiei. După realizarea decompozitiei, aplicaţia poate fi simulată utilizînd un *master task* care va fi înlocuit mai tirziu prin implementări specifice Scopul acestei simulări iniţiale este de a detecta acele taskuri a caror comportare afectează în cel mai mare grad performanţele intregului sistem. Utilizatorii pot experimenta diverse alternative de specificaţii comportamentale ale taskurilor pînă se obţin rezultatele impuse. In acest moment, se poate trece la înlocuirea descrierilor acestor taskuri generale cu descrieri mai detaliate cuprinzînd alte taskuri interne şi cozi, utilizînd facilităţile de descrire ale taskurilor puse la dispoziţie de Durra. Procesul de simulare continuă, luindu-se în considerare noile taskuri şi continuînd decompoziţia pînă la obţinerea unei structuri interne satisfăcătoare şi a unei comportări corecte a aplicaţiei, pe ansamblu, fiind posibile şi reluar de tip *backtracking*. Codificarea taskurilor se realizează numai după ce s-au definitivat specificaţiile lor de descriere acestea s-au dovedit a fi acceptabile şi nu mai este necesară continuarea decompoziţiei.

2.Lee şi Goldwasser sunt autorii limbajului *DICON*, destinat controlului execuţiei concurente a programelor secventiale scrise în limbajele C şi PROLOG.[LG85]

3.Magee şi Kramer sunt autorii limbajului *CONIC* destinat reconfigurării dinamice a unui sistem în timp real. CONIC restrictionează însă programarea taskurilor la o extensie a limbajului Pascal care include primitive pentru transferu mesajelor [MK83].

4.Belzile propune limbajul *RNET* pentru constructia aplicaţiilor distribuite în timp-real. Un program RNET const într-o specificatie a configuraţiei şi un cod procedural care sunt compilate şi legate cu un nucleu pentru execuţie. Sunt permise facilităţi legate de execuţia în timp-real precum specificarea unor timpi limită şi a unor întîrzieri folosite pentru monitorizarea şi planificarea proceselor [Bel89].

BUPT

1.2.3.1. Proiectarea unui suport pentru conectarea aplicaţiilor în mediu distribuit

O aplicaţie distribuită în cadrul sistemului proiectat [Mus96e][BDW89] TDL, constă într-un set de taskuri care comunică prin cozi de mesaje (fig. 1.2.3.2.1.). Descrierile taskurilor şi declaraţiile de tipuri specifică modul în care taskurile sunt lansate în execuţie şi executate ca procese concurente, tipul datelor prin care comunică procesele, precum şi cozile de date necesare proceselor producătoare şi consumatoare. Rezultatul unei compilări a unei descrieri de aplicaţie în limbajul TDL este un set de directive de alocare resurse şi de planificare.

Există trei faze distincte în dezvoltarea unei aplicaţii distribuite utilizînd limbajul: crearea unei biblioteci de taskuri, crearea unei aplicaţii utilizînd biblioteca de taskuri şi execuţia aplicaţiei (fig 1.2.3.2.2.)

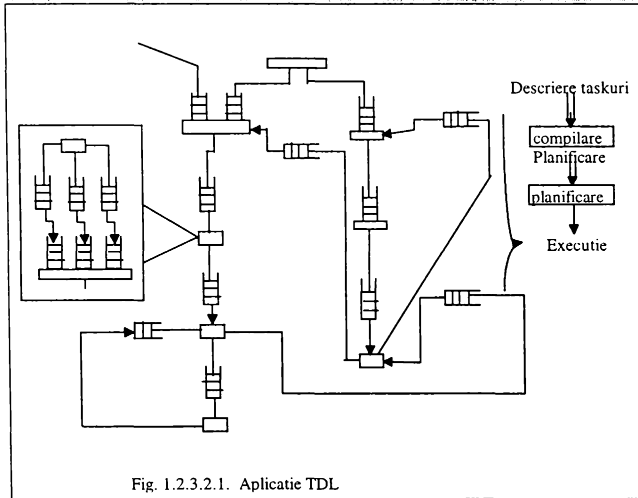


Fig. 1.2.3.2.1. Aplicatie TDL

În prima faza, folosind limbaje de programare adecvate, se scriu diversele taskuri care vor fi executate concurent în mediul eterogen. Pentru fiecare task implementat se scrie şi o descriere a taskului în limbajul TDL; pentru un anume task pot exista mai multe implementări, depinzînd de limbajul de programare folosit, tipul procesorului, caracteristici de performanţă şi alte proprietăţi. Fiecare implementare a unui task este însoţită de o descriere a taskului care este compilată şi catalogată în biblioteca.

Descrierea unui task include specificatii referitoare la performanţele implementării, tipuri de date pe care taskul le produce sau consumă, porturile utilizate pentru comunicaţia cu alte taskuri şi alte atribute statistice ale implementării.

A doua fază constă în crearea descrierii aplicaţiei. Din punct de vedere sintactic, descrierea aplicaţiei este o descriere a unui task şi poate fi catalogată în biblioteca la fel ca un task obisnuit. Aceasta permite scrierea unor ierarhii de descrieri de taskuri. După analiza descrierii aplicaţiei, analizorul generează un set de comenzi de alocare a resurselor şi de planificare a execuţiei, comenzi care vor fi interpretate de planificator. Ultima fază constă în execuţia aplicaţiei. Planificatorul încarcă implementările taskurilor în nodurile reţelei şi execută comenzile.

Descririle de taskuri, de fapt entităţile sintactice utilizate în constructia aplicaţiei, contin următoarele tipuri de informaţii: (1).porturile de interfaţă cu alte taskuri ; (2). atributele taskului. In figura 1.2.3.2.3. se prezintă structura generală a descrierii unui task.

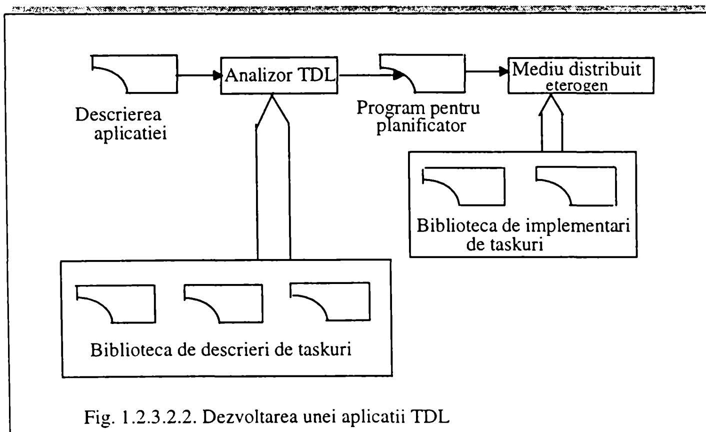


Fig. 1.2.3.2.2. Dezvoltarea unei aplicatii TDL

```
task nume_task
   ports          # pentru comunicatia dintre  un proces [i o coada
     declaratii de porturi
   attributes     # pentru a specifica proprietati ale taskului
     atribute valoare
  end  nume_task
```

Fig.1.2.3.2.3. Structura unei descrieri de task

BUPT

Partea de interfață cu alte taskuri conține descrierea *porturilor* de interfață specificînd tipul datelor care se transferă în/din port și direcția, astfel:

```
ports
     in1:in string;
     out1,out2:out int;
```

Partea de atribute specifică diverse informații precum: autor, numar de versiune, limbaj de programare, nume fișier, tip procesor sub forma:

```
attribute_
         author="xxxx";
         implementation ="nume_program";
         queue-size=25;
```

Folosirea părții de atribute pentru a selecta un task din biblioteca de taskuri este ilustrată în fig. 1.2.3.2.4..

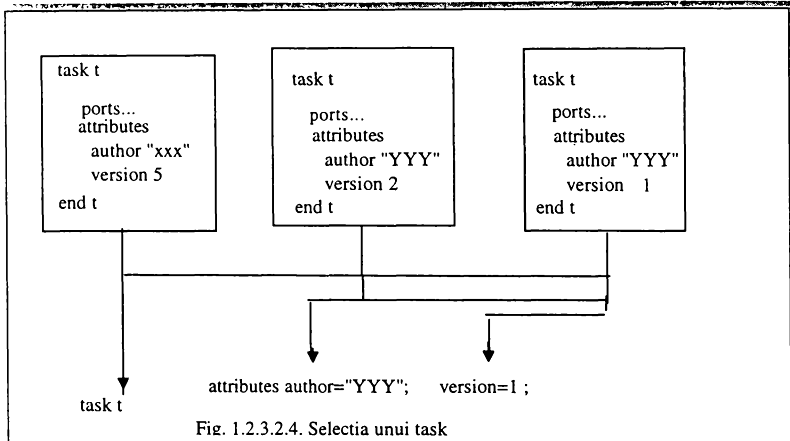


Fig. 1.2.3.2.4. Selectia unui task

Descrierea unei aplicații se face precizînd takurile componente și cozile prin care ele comunica. Există două tipuri de declaratii într-o aplicație: declaratiile din secțiunea de procese, introdusă de cuvîntul cheie *process și* declaratiile din sectiunea de cozi, introdusă de cuvantul cheie *queues:*

1. O declaratie de forma:

```
nume-proces: task nume_task;_
```

crează un exemplar al taskului selectat, ca un proces nou.

2. O declaratie de forma:

```
nume-coada[dim_coada]: nume-port-1>transformari-date>nume-port-2
```

crează o coadă în care datele se acumulează de la un port de ieșire al unui proces (*nume-port-1*) către un port de intrare al unui alt proces (*nume-port-2*) și în care datele sunt transferate în conformitate cu *transformari-date.*

Există trei componente active în mediul de execuție : *taskurile de aplicații, serverul și planificatorul* . In figura 1.2.3.2.5.se prezintă relatia dintre aceste componente.

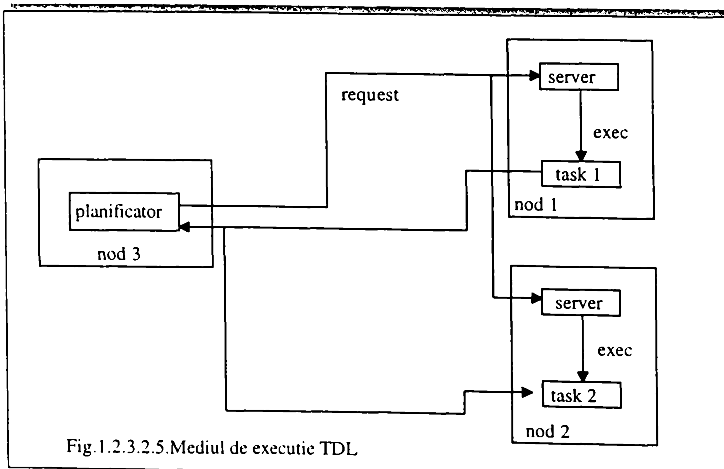


Fig.1.2.3.2.5.Mediul de executie TDL

BUPT

După analiza declarațiilor de tip, a descrierii taskurilor și a descrierii aplicației, se poate trece la execuția aplicației prin lansarea în fiecare nod a unui proces server și într-un nod separat al planificatorului. Planificatorul, care este apelat primind drept argument numele fișierului rezultat în urma analizei, are rolul de a starta taskurile, de a stabili liniile de comunicație și de a monitoriza execuția aplicației. In plus, implementează taskuri predefinite precum: *broadcast* și *merge* . In urma fazei de analiză se obțin instructiuni pentru planificator care descriu modul în care se execută procesele concurente, porturile și cozile utilizate pentru comunicație între procese tipurile de date ce se transferă între procese și posibilele reconfigurari ale aplicației. Serverul are rolul de a lansa taskul în nodul respectiv, la indicatia planificatorului.

In orice nod al rețelei în care se execută un task,se activează și un exemplar de server. .

In momentul lansării unui task, planificatorul ii furnizează, via server, informații precum: adresa nodului pe care se execută planificatorul, socket-ul Unix ce va fi folosit pentru comunicație task-planificator, și un numar întreg, identificator al taskului. Taskurile de aplicație utilizează interfața pentru a comunica între ele.

Taskurile de aplicație, în mod tipic, realizează următoarele operații: (1). Stabilesc comunicația cu planificatorul; (2).Cer identificatorii porturilor.; (3).Trimit și recepționează date; (4).Termină comunicația cu planificatorul.

Implementarea actuală (prima versiune) a fost realizată folosind protocolul TCP/IP în rețele Unix; se preconizează ca versiunea a 2-a să se extindă la rețele mixte Unix și MS Windows (Win 32).

Se prezintă, ca exemplu, în figurile 1.2.3.2.6. și 1.2.3.2.7. un program pentru următoarea aplicație simpla: taskul A,un task producator generează un sir de date de tip *string* către un task de tip predefinit *broadcasting* care trimite mai departe aceste siruri către alte două taskuri consumatoare identice.

```
task taskA;
  attributes
     processor="xantipa";
     implementation="string_producer";
  ports
    out1:out string;
end taskA;

task taskB;
  attributes
     processor="alpha";
     implementation="string_consumer";
  ports
    in1:in string;
end taskB;
```

Fig.1.2.3.2.6..Descrierile taskurilor și a tipurilor de date

```
task main
  structure
   process
    p1:task taskA;
    p2:task taskB;
    p3:task taskB;
    pb:task broadcast
         ports
           in1:string;
           out1,out2:out string;
         end broadcast;
   queues
      qb1:p1.out1> >pb.in1;
      qb2:pb.out2> >p2.in1;
      qb3:pb.out2> >p3.in1;
end main
```

Fig.1.2.3.2.7..Descrierea aplicației

BUPT

1.2.3.2. OSF DCE - Strategie şi arhitectură

Mediul de programare distribuită DCE (Distributed Computer Environement) [Tan96][BPK91][BGH93], creat la initiativa organizatiei OSF (Open Software Foundation) este o colecţie de componente software integrate în sistemul de operare UNIX; scopul DCE este de a furniza un suport pentru dezvoltarea şi execuţia aplicaţiilor distribuite în medii distribuite omogene. OSF DCE este inclus în mediul de prelucrare *deschisă* propus de OSF, alaturi de sistemul de operare Unix OSF/1, interfaţa grafică OSF/Motif şi OSF DME (Distributed Management Environement). Din punctul de vedere al sistemelor distribuite, DEC şi DME sunt de cea mai mare importanţă. In timp ce DCE constituie baza dezvoltării aplicaţiilor distribuite şi oferă servicii direct utilizatorului, DME abordează problemele gestionării reţelei şi sistemului, alaturi de suportul pentru protocolul SNMP. DCE tinde să devină astazi un standard industrial pentru prelucrarea distribuita.

Arhitectura generală a mediului DCE este reprezentată în fig. 1.2.3.3.1. Toate componentele se bazează pe serviciile sistemului de operare Unix şi pe serviciile de comunicaţie TCP/IP. Aplicaţiile distribuite dispun de serviciile fundamentale DCE via interfaţa de programare a limbajului C; celelalte servicii DCE sunt furnizate implicit prin intermediul serviciilor fundamentale sau prin intermediul serviciilor modificate ale sistemului de operare.

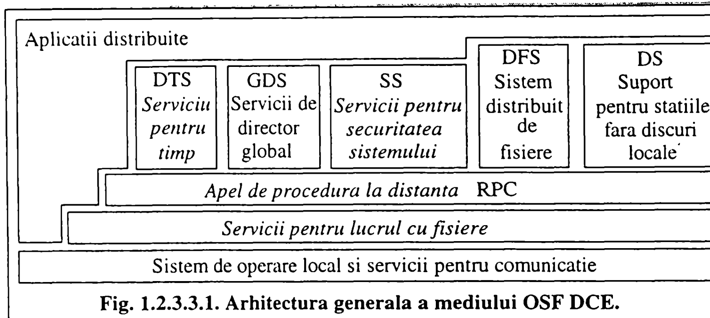


Fig. 1.2.3.3.1. Arhitectura generala a mediului OSF DCE.

Serviciile fundamentale DCE sunt:

(1) *Serviciile pentru lucrul cu fire*, care reprezintă o implementare portabilă a proceselor usoare (fire) în conformitate cu standardul POSIX 1003.4a. Firele fac posibilă prelucrarea concurentă într-un spaţiu de adrese partajat (vezi subcapitolul 1.3.2.1.) şi sunt utilizate în special de RPC pentru implementarea serverelor multifir şi pentru implementarea apelurilor asincrone de proceduri, fără blocare;

(2) *DCE RPC* ese suportul pentru comunicaţiile între sisteme eterogene. O cerere pentru serviciu a unui client se transmite printr-un apel de procedură obişnuit, confirmat prin primirea rezultatelor de la server. Conversia de parametrii şi transmiterea efectivă de mesaje se realizează prin intermediul componentelor denumite *RPC stub*, aflate atît în mediul clientului cît şi al serverului (DCE RPC este descris în detaliu în subcapitolul 3.4.9.A); acestea se generează automat plecînd de la o scurtă descriere a procedurilor (asemănătoare prototipurilor funcţiilor din limbajul C). DCE oferă în acest scop limbajul IDL (Interface Definition Language). In plus, DCE mai oferă şi posibilitatea mai multor tipuri de semantică a apelurilor, securitatea apelurilor de proceduri, cu autentificare şi autorizare bazate pe serviciul de securitate DCE, posibilitatea utilizării unor nume globale de servere, pe baza standardului de servicii directoare X.500;

(3) *CDS (Cell Directory Service)* asigură gestiunea numelor în cadrul unui domeniu. Aceasta constă în maparea numelor în adrese şi actualizarea informaţiilor după nume; în plus, CDS stă la baza mapării adreselor de servere în cererile clienţilor RPC. CDS utilizează pentru creşterea eficienţei replicarea şi cache-ingul. O interfaţă de programare avansată este oferită prin X/Open Directory Service Interface;

(4) *Serviciul de securitate* implementează autentificarea, autorizarea şi criptarea. Aceste mecanisme sunt puternic integrate în DCE RPC; de exemplu, clienţii şi serverele se pot autentifică reciproc, serverele pot verifică dinamic liste de control ale accesului pentru autorizarea clienţilor, iar mesajele RPC pot fi criptate la cerere;

(5) *DTS (Distributed Time Service)* implementează sincronizarea ceasurilor în mediul distribuit (o problemă comună tuturor mediilor distribuite, vezi subcapitolul 4.3.1.); se garantează că ceasurile locale ale nodurilor participante sunt sincronizate în interiorul unui interval si, eventual, cu o sursă externă exactă. Acest serviciu este impotrant pentru implementarea algoritmilor distribuiti, care se bazează pe marca de timp; este, de esemenea, folosit şi de alte componente DCE.

Alte servicii DCE sunt: (1) *GDS (Global Directory Service)* care extinde CDS-ul, furnizînd facilităţile de nume în afara domeniilor; se bazează pe standardul X.500. Ca urmare, face posibilă cooperarea nu numai cu servere DCE, dar şi cu alte servere X.500. Este prevazută şi alternativa Internet Domain Name Service pentru nume globale; (2) *DFS (Distributed File System*) implementează gestiunea fişierelor într-o manieră distribuită. Fisierele pot fi localizate pe diverse servere şi pot fi replicate; programele client pot lucra cu fişierele independent de localizarea lor, ca şi cu sistemul local de fişiere Unix. Accesul la fişiere este optimizat prin mecanismul de cache la nivel de fişier în nodurile clienţilor. Interoperatibilitatea cu Network File System este asigurată printr-o interfaţă NFS/DFS; în plus, printr-o componenetă specială (Diskless Support, DS) se asigură funcţiile de boot, swap şi lucru cu fişiere şi pentru staţiile fără disc local.

DCE furnizează structurarea sistemelor distribuite pe celule, cu scopul de a mentine în limite rezonabile dimensiunea domeniilor administrative. O celulă cuprinde nodurile atasate unei reţele locale dar, în mod obisnuit, se defineste în funcţie de consideratii de organizare şi mai puţin în funcţie de structura reţelei fizice; ca urmare, o celulă apare ca un set de noduri gestionate de o autoritate comuna. Multe din serviciile DCE sunt optimizate special pentru accesul intra-celula; traversarea graniţei celulei, către servere din afara celulei impune autentificare specială şi

BUPT

proceduri de autorizare diferite de cele pentru accesul intra-celula. Sistemul distribuit de fişiere oferă localizarea total transparentă pentru accesul intra-celula; la traversarea celulelor se cere însă specificarea explicită a numelor de celule.

In final se prezintă în fig. 1.2.3.3.2. un exemplu de aplicaţie distribuită în mediul OSF DCE. Comunicarea între procesele aplicaţiei se prezintă în fig. 1.2.3.3.3.; aplicaţia constă în controlul productiei şi vanzarilor unei firme. Fiecare celulă contine un set de servere (DTS, DFS, CDS, SS); comunicaţia intra-celula şi inter-celule se realizează via RPC.

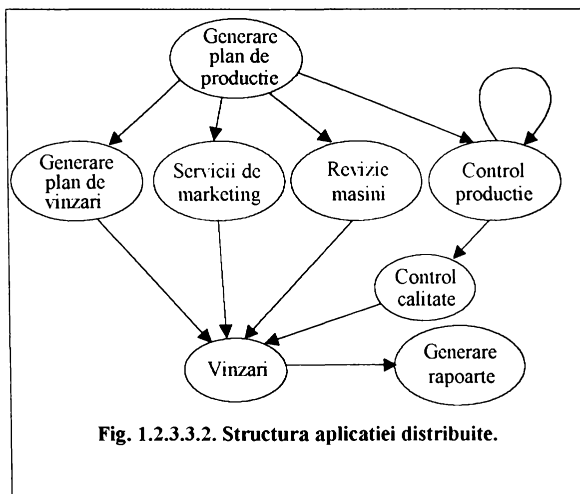


Fig. 1.2.3.3.2. Structura aplicatiei distribuite.

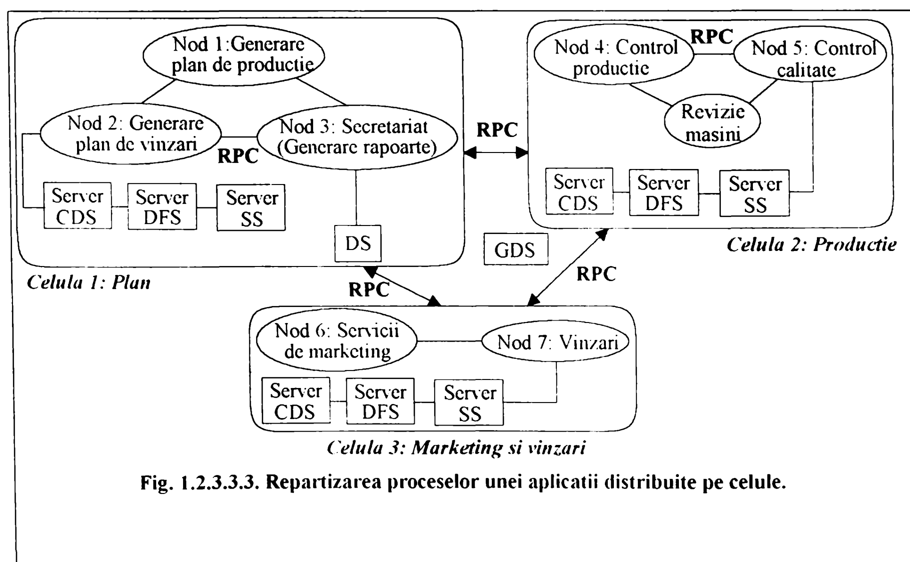


Fig. 1.2.3.3.3. Repartizarea proceselor unei aplicatii distribuite pe celule.

BUPT

1.3. Sisteme de operare pentru rețele de calculatoare și sisteme de operare distribuite

Intr-o rețea de calculatoare o resursă ce trebuie gestionată este însăși rețeaua (comunicațiile în rețea). In arhitectura stratificată a rețelelor propusă de modelul de referință OSI, fiecare nivel dintr-un nod al rețelei va avea de implementat protocolul sau specific. Aceste protocoale apar astfel ca o prelungire a sistemului de operare în fiecare nod al rețelei. Pe de altă parte, unul din scopurile proiectării rețelelor este acela de a oferi utilizatorilor și servicii situate la distanță, localizate în rețea și nu în nodul de proveniență a cererii. Ramîne în sarcina sistemului de operare să identifice astfel de cereri, să le localizeze și să detecteze accesele neautorizate la serviciile oferite restrictiv în rețea (implementînd astfel un mecanism de validare a cererilor). Deoarece într-o rețea de calculatoare pot fi legate, în general, calculatoare de diverse tipuri, provenind de la firme producatoare diferite, este posibil ca fiecare nod să fie dotat cu propriul sau sistem de operare. Aceste sisteme de operare locale, vor gestiona resursele locale, ale nodului respectiv, fără a avea cunostinte despre rețeaua în întregime. Pentru a beneficia de serviciile rețelei există în această situație două soluții: o prima soluție este de a adăuga fiecarui sisrem de operare local un proces nou care va deservi cererile pentru acces la serviciile din rețea; altă soluție este de a proiecta un alt sistem de operare, care va înlocui sistemele de operare din toate nodurile. Procesul cu rol de interfață adaugat sistemului de operare în fiecare nod al rețelei se numește *proces agent*.Totalitatea proceselor agent din rețea formează *sistemul de operare al rețelei de calculatoare (NOS-network operating system)*. Folosind prima soluție, se extind astfel sistemele de operare existente; folosind a doua soluție, se obține un sistem de operare unic, care optimizează gestiunea tuturor resurselor din rețea *(DOS-distributed operating system)*. Cele două soluții, NOS și DOS au drept scop furnizarea de servicii aplicațiilor, servicii legate de gestiunea proceselor, gestiunea comunicațiilor, gestiunea perifericelor, gestiunea memoriei și gestiunea fișierelor. Diferența majoră dintre cele două soluții este în maniera de abordare a gestiunii acestor resurse. Modelul NOS tratează resursele fiecarui nod ca fiind locale acelui nod și accesibile prin cereri explicite la interventia proceselor agent. Modelul DOS tratează toate resursele distribuite în rețea într-o maniera globala, controlul și gestiunea tuturor acestor resurse realizîndu-se folosind strategii unice în toată rețeaua. De exemplu, dacă un utilizator initiaza execuția unui proces într-un calculator situat la distanță, în modelul NOS el trebuie să se conecteze la rețea, printr-o cerere explicită și să comunice nodului respectiv cererea sa. In sistemul de operare al calculatorului situat la distanță, procesul initiat va fi perceput ca un proces local și va fi gestionat în consecinta. In modelul DOS, initierea unui proces este lasată complet în sarcina sistemului de operare. Sistemul de operare examinează blocul de control al procesului pentru a determina eventualele cerinte specifice, dupa care planificatorul de procese, în funcție de încarcarea nodurilor ia în considerare procesul creat. Planificarea proceselor se face cu scopul de o optimiza gestiunea proceselor din întreaga rețea. Aceasta implică o transparență totală față de utilizatori care percep rețeaua ca un tot unitar și nu ca o colectie de dispozitive autonome care cooperează. Tipic, un sistem de operare distribuit este compus dintr-un nucleu, ale carui copii sunt dispuse în toate nodurile și dintr-o ierarhie de componente software pentru gestiunea resurselor, situate deasupra acestui nucleu. Aceste componente pot fi organizate urmind modelele calculului distribuit (deoarece sistemul de operare este un exemplu de prelucrare distribuita): singular, federativ, replicat sau factorizat. In fig.1.3.1. este prezentată comparativ organizarea celor două tipuri de sisteme de operare pentru rețea: NOS și DOS [Mus93d][Cri89][Tan96].

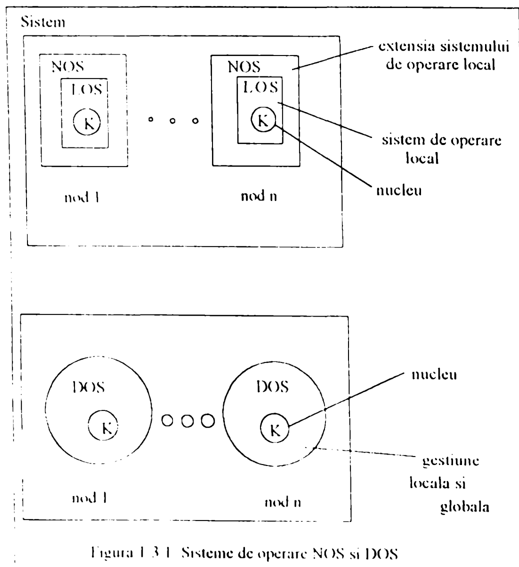


Figura 1.3.1. Sisteme de operare NOS si DOS

Diferența dintre sistemele de operare distribuite și cele pentru rețelele de calculatoare poate fi reliefată sugestiv dacă se considera nivelul pe care il ocupa comunicația între procese (IPC) în arhitectura sistemului de operare (fig. 1.3.2.): cu cît IPC este situat la nivele mai inferioare, cu atît crește transparența localizării resurselor în rețea. Astfel, in fig. 1.3.2.c, dacă IPC este situat chiar in nivelul de gestionare al proceselor se obține maximum de transparență și sunt satisfacute astfel cerintele unui sistem de operare distribuit.

Spre deosebire de sistemele centralizate, controlul in sistemele de operare distribuite se realizează pe baza de negociere, consens și compromis între diversele componente raspindite în rețea. Intr-un sistem de operare centralizat deciziile care se iau in gestiunea resurselor, se bazează pe o cunoastere exhaustiva, în orice moment a stării curente a sistemului; in contrast, sistemul de operare distribuit își bazează deciziile pe o apreciere, niciodată exhaustiva a stării rețelei, pe informații statistice și procentuale și foloseste în acest scop metode și tehnici ca metoda *divide et impera*, teoria deciziilor, teoria grafurilor, teoria jocurilor, algoritmi de control al concurenței în bazele de date.

BUPT

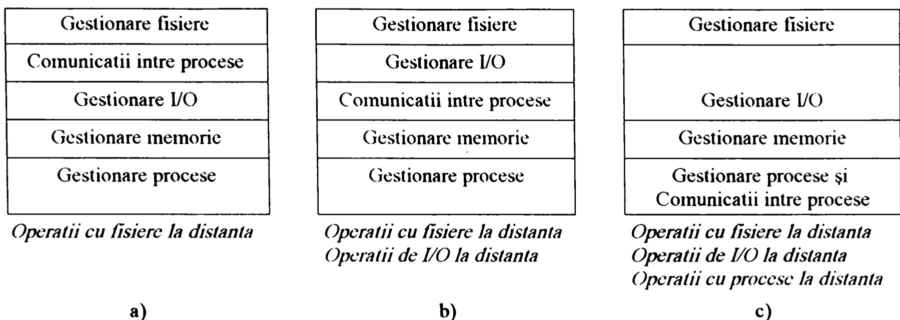


Fig.1.3.2. Nivele de transparenta obtinute în funcţie de plasarea nivelului de comunicatie între procese

1.3.1.Specificul sistemelor de operare pentru reţele de calculatoare

O caracteristică importantă a reţelelor de calculatoare, în special a reţelelor generalizate este posibilitatea conectării în reţea a diverse tipuri de calculatoare. Apare astfel necesară proiectarea sistemului de operare pentru reţeaua de calculatoare, sistem care să permită însă execuţia sistemelor de operare locale, în nodurile reţelei. El trebuie să contină limbajul de comandă prin care utilizatorii să-şi prezinte cererile în reţea şi procesele agent prin care aceste cereri sunt acceptate, transmise şi recunoscute în reţea.

1.3.1.1. Limbajul de comandă

Instrucţiunile limbajului de comandă sunt cereri de servicii adresate sistemului de operare de către utilizatori. Ele sunt tratate de interpretorul de comenzi care se constituite fie ca nivelul superior al sistemului de operare fie ca un proces separat, distinct faţă de sistemul de operare. Limbajul de comandă al sistemului de operare pentru reţea va cuprinde setul de cereri de servicii posibile în reţea prin care utilizatorii au acces la facilităţile oferite de reţea. Comenzile trebuie să fie, pe de o parte, distincte faţă de comenzile sistemului de operare local, dar pe de altă parte trebuie să refere accesul la distanţă într-o manieră cît mai transparentă pentru utilizator. De exemplu, un transfer de fişiere între două noduri poate fi vazut ca un caz particular al unei comenzi de tip *copy:* distinctia între accesul local şi accesul la distanţă se face prin argumentele comenzii. Dacă nivelul de transparenţă este mai scazut, într-o comandă de tipul "*copy* sursă destinaţie", a cărei semnificatie este copiază fişierul "sursă" situat la distanţă în fişierul local "destinaţie", se poate specifică explicit localizarea fişierului "sursă" prin nume de cale, de exemplu "nod/nume-utilizator/sursă". Dacă, dimpotrivă, nivelul de transparenţă este mare, în cîmpul "sursă" se va specifica numai numele fişierului, fără localizare. Ramîne în sarcina sistemului de operare să gasească adresa nodului care conţine fişierul ceea ce implică pastrarea unor structuri de date în care să se memoreze corespondenţa dintre numele locale ale fişierelor şi localizarea lor. Sistemul de operare al reţelei poate recunoaste şi apeluri sistem adresate reţelei în timpul execuţiei unui program. De exemplu apelul sistem: *fork(nod1/nume-utilizator/nume-fişier,nod2)* poate fi interpretat ca avînd următoarea semnificaţie: cu fişierul */nume-utilizator/nume-fişier* conţinînd forma executabilă a unui program din nodul "nod1", se crează şi execută un proces în nodul "nod2". Responsabilitatea sistemului de operare al reţelei este să capteze astfel de cereri care implică accesul la distanţă, să le interpreteze şi să le transmită mai departe nivelelor inferioare de protocol.

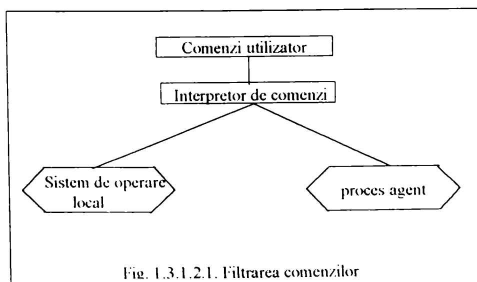


Fig. 1.3.1.2.1. Filtrarea comenzilor

	nod	login	cont	acces
Utilizator 1 →	UKC	IMC.OW 1	179600	mail
Utilizator 2				

Fig 1.3.1.2.2.Structuri de date utilizate pentru conectarea la distant

BUPT

1.3.1.2. Procesele agent.

Intr-o rețea de calculatoare cu sisteme de operare eterogene, fiecare nod se conectează la rețea prin intermediul unui *proces agent*. Comenzile cu efect local sunt separate şi tratate de sistemul de operare local; cele ce implică acces la distanță sunt transmise procesului agent (fig. 1.3.1.2.1.), pentru a putea le transmite mai departe către nodul care le poate indeplini. Procesul agent mentine structuri de date care ii permit să facă legatura între cele trei elemente ale comenzii: utilizator, nod la distanță, serviciu. Pentru fiecare utilizator trebuie să existe o indicație relativă la nodurile la care se poate conecta: parole, conturi folosite la conectare, tipurile de servicii permise (fig.1.3.1.2.2).

O categorie importantă de servicii oferite utilizatorilor de către sistemul de operare al rețelei se referă la manipularea fişierelor. Sistemul de operare trebuie să furnizeze utilizatorilor facilitatea de a accesa fişiere la distanță, cu verificarea autenticității cererilor, într-o manieră mai mult sau mai puțin transparentă. Dacă nivelul de transparență este mic, utilizatorul va indica şi denumirea simbolică a nodului la distanță implicat în transfer; sarcina procesului agent este doar de a determina adresa fizică a acestui nod, ramînînd în sarcina sistemului de operare local nodului să determine fişierul implicat în transfer prin analiza numelui de cale al fişierului furnizat de procesul agent. Dacă nivelul de transparență în localizarea fişierelor este mare, procesul agent mentine un catalog al fişierelor accesibile - via proces agent - în rețea. Utilizatorii nu trebuie să cunoască localizarea acestor fişiere în rețea. Mai mult,dacă procesul agent mentine pentru fiecare utilizator un catalog cu fişierele accesibile în rețea, el poate fi realizat astfel încît să ascundă numele fişierelor şi alte conventii locale față de utilizatorii situați la distanță. Procesul agent va pastra o baza de date care memorează corespondența nume virtual al fişierului-nume nod, nume fişier. (fig. 1.3.1.2.3.).

Pentru implementarea procesului agent trebuie avute în vedere două probleme: mecanismul interacțiunii cu sistemul de operare local şi localizarea procesului agent. Dacă sistemului de operare local ii este caracteristic interacțiunea între procese folosind apelul de procedură atunci şi comunicarea cu procesul agent se va face tot prin apel de procedura. Dacă tehnică de interacțiune între procese, folosită de sistemul de operare local se bazează pe transmiterea de mesaje atunci şi interacțiunea cu procesul agent se face cu ajutorul mesajelor.

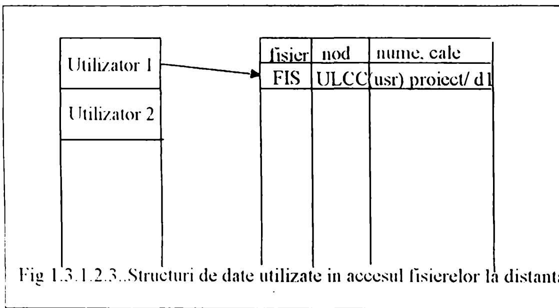


Fig 1.3.1.2.3..Structuri de date utilizate in accesul fisierelor la distant:

In ceea ce priveşte localizarea procesului agent există trei variante folosite: (1) procesul agent poate fi implementat ca un proces separat în fiecare nod al rețelei; (2) poate fi implementat ca o componentă a nucleului sistemului de operare local; (3) poate fi plasat într-un nod special;

In prima varianta, (fig.1.3.1.2.4.) toate comunicațiile dintre procesele utilizator şi procesul agent se fac prin intermediul nucleului sistemului de operare local. Aceasta inseamnă că pentru transmiterea unui mesaj în rețea, au loc trei interacțiuni utilizator-nucleu: prima interacțiune este apelul sistem al utilizatorului către procesul agent, a doua este transmiterea mesajului la procesul agent de către nucleu, iar a treia este transmiterea de către agent a mesajului în rețea, transmitere care se face folosind protocoalele din nivelele inferioare ale rețelei, implementate in nucleul sistemului de operare. Această triplă interacțiune poate conduce la un timp de răspuns uneori prea mare. Situația poate fi şi mai dezavantajoasă dacă procesul agent este tratat ca orice proces utilizator şi evacuat din memoria internă. Totuşi, această variantă are avantajul că nu necesită creşterea dimensiunii nucleului.

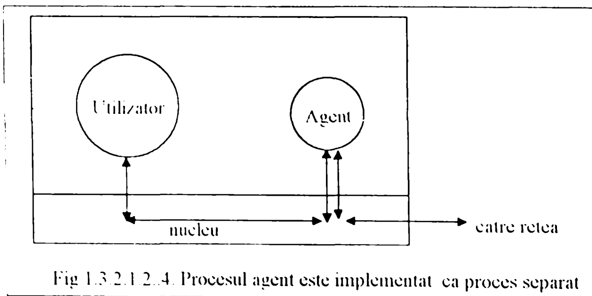


Fig 1.3.2.1.2..4. Procesul agent este implementat ca proces separat

Varianta a doua (fig. 1.3.1.2.5) evită traficul suplimentar de mesaje generat in prima variantă dar dimensiunea nucleului este marită considerabil; această creştere a dimensiunii poate fi uneori intolerabilă.

In varianta a treia (fig. 1.3.1.2.6.) agentul este rezident intr-un nod special.

El constă de fapt într-o colecție de module, cite unul pentru fiecare sistem de operare local al fiecarui nod. Din aceste noduri, accesul la rețea se face numai prin

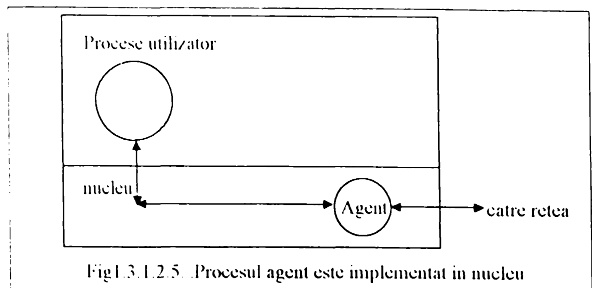


Fig1.3.1.2.5. .Procesul agent este implementat in nucleu

BUPT

intermediul nodului agentului; de aceea sunt necesare rutine de protocol la nivelele inferioare ale reţelei care să asigure emisia şi recepţia mesajelor de la nodul agent. Metoda are dezavantajul că este vulnerabilă la defectiunile nodului agent dar şi avantajul că simplifică implementarea. Ea se aplică numai în situaţia în care agentul asigură un anumit tip de servicii în reţea (gestiunea fişierelor); în aceste situaţii pot exista mai multi agenti, fiecare asigurind un anumit tip de servicii. Această variantă poate fi clasificată în modelul general *client-servant.*

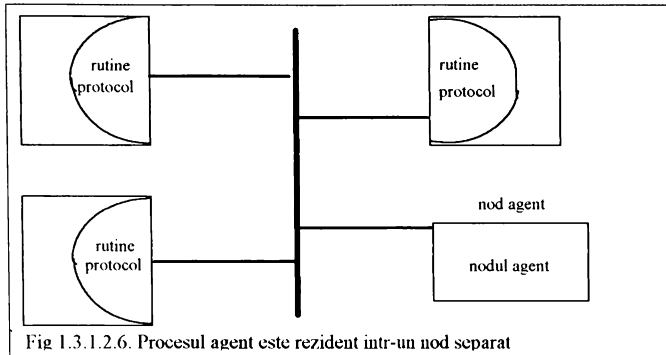


Fig 1.3.1.2.6. Procesul agent este rezident intr-un nod separat

1.3.2 Specificul sistemelor de operare distribuită

In acest subcapitol se prezintă probleme legate de gestionarea procesoarelor şi a fişierelor într-un sistem de operare distribuit. Problemele legate de comunicaţie, interacţiunea între procese şi sincronizarea proceselor într-un sistem distribuit sunt prezentate în capitolele următoare, 2, 3 şi 4.

1.3.2.1. Gestionarea proceselor

Intr-un sistem de operare distribuit, funcţia de gestionare a proceselor trebuie să furnizeze strategii şi mecanisme pentru: (1) operaţii asupra proceselor, atît a proceselor locale, cît şi a celor de la distanţă: crearea, distrugerea, terminarea, suspendarea, modificarea atributelor, modificarea stării proceselor; (2) execuţia concurentă a proceselor; (3) planificarea pentru execuţie a proceselor; (4) lansarea proceselor în execuţie pe procesoare la distanţă; (5) migrarea proceselor; (6) încarcarea echilibrată a procesoarelor[Mus93d].

Faţă de sistemele centralizate, în sistemele distribuite, funcţia de gestionare a proceselor are un handicap în plus: ea trebuie realizată în condiţiile unor informaţii de stare incomplete, deoarece, în astfel de sisteme, mesajele, utilizate pentru comunicarea între procese a informaţiilor de stare sunt întîrziate (uneori chiar pierdute); starea transmisă prin mesaje este intotdeauna starea dintr-un moment de timp anterior şi nu din momentul curent. In plus, în sistemele distribuite, pentru a se imbunatăţi timpul de răspuns, încărcarea procesoarelor şi partajarea resurselor, sunt necesare noi tipuri de operaţii cu procesele: lansarea lor în execuţie pe un procesor de la distanţă şi migrarea proceselor.

Există două tipuri de unităţi de lucru pentru procesor furnizate de sistemele de operare distribuite ([Gos91] [Tan93] [Tan96]): ***procese grele*** şi ***procese usoare*** sau ***fire***. Diviziunea proceselor în cele două categorii se bazează pe gradul de partajare al memoriei. Astfel, *procesele grele*, cărora sistemul de operare le asociază blocuri de control sau descriptori de proces au spatii de adresa proprii, atît pentru cod, cît şi pentru date şi stiva, dar şi tabelă de descriptori fişiere, timere, semnale, semafoare proprii. *Procesele usoare* partajează datele şi au numai stivă proprie (fig. 1.3.2.1.); ele partajează doar variabile locale, fişiere deschise, timere, semnale.

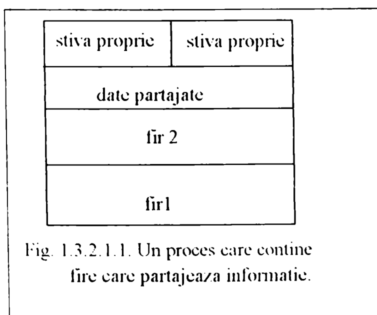


Fig. 1.3.2.1.1. Un proces care contine fire care partajeaza informatie.

Firele au fost introduse pentru: (1) a se putea realiza execuţia concurentă şi în cazul apelurilor sistem cu blocare; (2) a se reduce regia de sistem datorată comutării contextului proceselor; (3) a se asigură tratarea concurentă a unor acţiuni ale utilizatorilor (de exemplu, în sistemul de gestiune al ferestrelor); (4) a se partaja informaţii în cadrul programelor utilizator. Deoarece firele partajează memoria, accesul la datele comune se face în regim de excludere reciproca. Pentru sincronizarea firelor (inclusiv în accesul la sectiunile critice) se folosesc în general semafoare binare (numite *mutex*-uri) şi *variabile condiţie* (asemanatoare variabilelor condiţie folosite pentru sincronizarea cu monitoare); fiecare variabilă condiţie, cînd este creata, se asociază unui *mutex*. Diferenţa dintre mutex-uri şi variabilele condiţie este aceea că mutex-urile se folosesc pentru blocarea pe termen scurt (pentru blocarea accesului la o sectiune critica), în timp ce variabilele condiţie se utilizează pentru blocarea

BUPT

pe termen lung (pentru a se astepta disponibilitatea unor resuse). Evitarea interblocării firelor se face prin utilizarea combinată a mutex-urilor şi a variabilelor condiţie (fig. 1.3.2.1.2.)([Tan93],[Tan96]). Variabila condiţie se utilizează pentru a achizitiona resurse, deoarece la efectuarea unei operaţii *wait* asupra ei se eliberează automat şi atomic mutex-ul asociat (1.3.2.1.2.a). Cînd mai tîrziu, firul care deţine resursa, o eliberează, execută o operaţie *wake-up (variabila_condi]ie),* deblochează firul (sau toate firele) care aşteaptă la condiţie şi mutex-ul este alocat din nou.

```
lock (mutex)
      *verifica structurile de date
      while (*resursa ocupata)
                wait (variabila_conditie)
      *marcheaza_resursa_ocupata
unlock (mutex)
      (a)

lock (mutex)
      marcheaza_resursa_libera
unlock (mutex)
wakeup (variabila_conditie)
      (b)
```

Fig. 1.3.2.1.2. Utilizarea combinată a mutex-urilor şi a variabilelor conditie

Se apreciază ([Gos91]) că puţine sisteme de operare distribuite permit lucrul cu fire şi deci partajarea memoriei între unităţi de lucru ale aceluiaşi proces. Astfel, sistemul Charlotte, creat la Universitatea Wisconsin-Madison, de către o echipă de cercetatori condusă de Finkel ([Fin89]) este un exemplu de sistem care nu permite partajarea memoriei prin intermediul firelor; în schimb, sistemul V, creat la Universitatea Stanford, de către o echipă de cercetatori condusă de Cheriton ([Che88]) reuneşte toate procesele cu acelaşi spaţiu de adrese sub numele de *echipă de procese*; astfel, serverele pot utiliza o zonă de memorie comuna, care implementează de obicei buffere. Sistemul Mach, creat la Universitatea Carnegie-Mellon de către o echipă de cercetatori condusă de Rashid, permite lucrul cu task-uri şi fire în cadrul task-urilor; în Mach este tipic un numar mic de taskuri şi multe fire în cadrul unui task. Mach permite partajarea memoriei între task-uri, atît sub forma *read/write* cît şi sub forma *copy-on-write*. Taskul este unitatea de alocare pentru resurse, atît în ceea ce priveşte spaţiul de adrese paginat, cît şi în ceea ce priveşte protecţia accesului la resurse; drept rezultat, fiecare task fiu partajează memoria cu taskul părinte în conformitate cu un atribut de mostenire.

Sistemele de operare distribuite trebuie să permită efectuarea de operaţii asupra proceselor la distanţă, într-un mod transparent pentru utilizator. Suportul pentru astfel de operaţii trebuie să includă primitive şi facilităţi pentru crearea, distrugerea, suspendarea şi continuarea unui proces, pentru alocarea/dealocarea de porturi pentru comunicaţie, ca şi pentru lansarea şi migrarea unui proces pe un alt procesor. Un model pentru crearea proceselor la distanţă este oferit de sistemul V [Che88]. Datorită facilităţilor de comunicaţie dintre procese, un proces poate fi creat, prin intermediul apelului sistem, *CreatProces*()si la distanţă; acesta construieste şi transmite un mesaj nucleului nodului destinaţie care creează procesul, asociindu-i un identificator (pid) unic. Noul proces creat ramane în starea *awaiting _reply* pînă la transmisia de către părinte a datelor iniţiale care permit începerea execuţiei.

În ceea ce priveşte *crearea proceselor* există mai multe implementari posibile: (1) execuţia părintelui se desfăşoară concurent cu cea a proceselor fii (sincronizarea execuţiei între aceştia se poate face cu construcţii de tipul fork/join) sau procesul părinte aşteaptă pînă se termină fii sai; (2)(a) procesele fii partajează numai un subset din datele parintelui sau toate variabilele în comun. Iată două exemple: (1) sistemul distribuit LOCUS, o extensie a sistemului Unix în mediu distribuit; in LOCUS, apelurile Unix *fork()* şi *exec()* au fost extinse astfel încît: *fork()* poate crea un proces la distanţă, cu acelaşi cod ca şi apelantul, dar cu stivă şi date separate iar *exec()* a fost extins pentru a permite unui proces să migreze pe alt nod. In plus au fost introduse două apeluri noi: *run()*, al carui efect logic este combinatia *fork()+exec()* şi care permite initierea execuţiei unui program fie local, fie la distanţă, şi *migrate()* care permite unui proces să-şi modifice nodul în care se executa; (2) sistemul Sprite, creează un nou proces, cu apelul *ProcFork()* care poate avea două comportamente, fie analog apelului *fork()* din Unix, fie poate cere noului proces să-şi partajeze datele cu procesul părinte (fig.1.3.2.3).

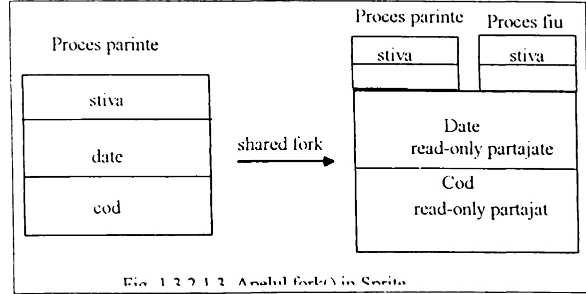


Fig. 1.3.2.1.3. Apelul fork() in Sprite

Un alt aspect legat de gestiunea proceselor este *crearea şi distrugerea porturilor*, în sistemele în care comunicaţia între procese se face prin porturi. Un port poate fi creat la crearea procesel sau la cererea explicită a proceselor. La crearea unui proces, procesul părinte poate asocia fiului un port de date şi un port de servicii; acestea vor fi utilizate pentru comunicaţie

BUPT

cu sistemul. O problemă care se pune însă este cum procesul începe comunicaţie cu alte procese: o soluţie este ca procesul fiu să comunice numai cu sistemul şi să ceară acestuia crearea de porturi cînd trebuie să comunice cu alte procese; altă soluţie este ca părintele să-i ofere fiului un set de porturi pe care acesta să le poată folosi imediat. Un port poate exista fie pînă la distrugerea procesului proprietar, fie pînă la distrugerea procesului. In mod obisnuit, numai creatorul portului poate primi mesaje în acel port; drepturile pentru emisie pot fi transmise de către creator la orice proces cu care se doreste comunicarea.

In ceea ce priveşte ***execuţia la distanţă a proceselor*** există următoarele aspecte: (1) nivelele la care trebuie furnizate facilităţi pentru execuţie la distanţă: (a) la nivelul programelor, în care instrumentul principal este apelul de proceduri la distanţă si, mai rar, particularităţi de partajare a memoriei; (b) la nivelul limbajului de comanda. (2) handicapul pe care il prezintă execuţia la distanţă în mediile eterogene: sunt necesare translatari datorită modificării arhitecturii hardware la trecerea pe un alt nod, dificultatea de a asigura un spaţiu de nume globale, dificultatea de a descrie serviciile de executat, datorită diferentelor sintactice şi semantice în liniile de comanda ale sistemelor diferite; (3) cerintele impuse unei execuţii la distanţă: (a) existenţa unui mecanism care să identifice nodurile slab încărcate; (b) posibilitatea ca utilizatorul de la un nod să poată stopa execuţia proceselor altui utilizator, care determină degradarea performanţelor proceselor proprii (preemptibilitate); (c) execuţia la distanţă să poată fi realizată cu rezultate comparative cu execuţia locală; (4) selecţia unui nod (server) pentru execuţia la distanţă a unui proces se face fie pe baza propagării informaţiilor despre un server disponibil sau a cererilor pentru servere a clienţilor, fie pe baza informaţiilor furnizate de sistemul de gestiune al resurselor. În primul caz, se utilizează o mică bază de date, locală fiecarui nod care contine date relative la serverele în care nodul este interesat şi date despre clienţii cărora le poate oferi resurse. Numai două operaţii sunt disponibile asupra acestei baze de date: adăugare şi ştergere; ambele sunt realizate în urma recepţiei unor mesaje broadcast transmise de un server care doreste să se facă disponibil. Pentru a selecta un server disponibil, un client foloseşte baza de date locală, eventual după actualizarea ei în urma transmiterii broadcast a cererii.

Migrarea proceselor într-un sistem distribuit este facilitatea care asigură relocarea dinamică a proceselor de pe un procesor pe alt procesor. Operatia poate fi initiată fără stirea procesului respectiv, şi a proceselor cu care acesta interacţionează. Migrarea proceselor este justificată prin următoarele cerinte: (1) repartizarea eficientă a proceselor pe procesoare; (2) creşterea gradului de paralelism ale operaţiilor; (3) execuţia proceselor cu cerinte speciale de resurse pe procesoarele care satisfac cel mai bine aceste cerinte; (4) execuţia proceselor care lucrează cu fişiere de dimensiuni mari pe nodurile care contin acele fişiere, pentru a reduce astfel traficul de mesaje în sistem; (5) creşterea sigurantei de funcţionare a sistemului. Dintre problemele numeroase pe care le ridică migrarea proceselor, se enumera: (1) determinarea contextului unui proces şi detaşarea procesului de acesta, transferul procesului şi conectarea sa într-un nou context (cod, date, stiva, registre. cozi de mesaje, semafoare); (2) tratarea mesajelor transmise procesului după migrarea sa; (3) asigurarea transparenţei migrării proceselor; acest lucru este uneori dificil de obţinut deoarece starea procesului poate fi distribuită în mai multe tabele ale sistemului de operare, şi deci dificil de extras şi apoi de creat în nodul destinaţie; (4) asigurarea unei strategii de migrare care defineste *cînd* are loc migrarea unui proces şi *unde* are loc, în scopul obţinerii unor performanţe superioare; (5) determinarea momentului în care procesul care migrează poate să-şi înceapă execuţia pe nodul destinaţie; această problemă este generată de necesitatea copierii întregului spaţiu de adresă al unui proces. O exceptie de la această regula o constituie sistemul Spice, creat la Universitatea Carnegie-Mellon ([Zay87]), care oferă posibilitatea ca un proces să-şi înceapă execuţia la destinaţie aproape imediat, utilizînd un transfer *copy-on-reference* (fig.1.3.21..4); (6) migrarea proceselor în medii eterogene necesită în afară de salvarea şi restaurarea stării contextului proceselor şi interpretarea consistentă a datelor. In asemenea situaţii se pun probleme de translatare a datelor; o soluţie a acestor probleme este folosirea unei reprezentari standard pentru transportul datelor.

In ceea ce priveşte tratarea mesajelor primite de un proces care urmează să migreze există trei soluţii: (1) *soluţia redirectării mesajelor*. In momentul în care un proces este suspendat pentru migrare, se memorează toate mesajele primite şi

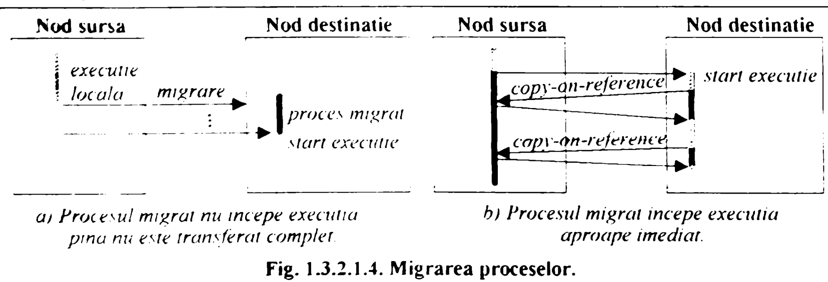


Fig. 1.3.2.1.4. Migrarea proceselor.

se trimit la noua destinaţie a procesului. după migrare; nu se informează procesele emiţătoare despre noua adresă a procesului migrat, astfel că nodul sursa va trebui să retransmită toate mesajele la destinaţie (nodul în care procesul a migrat); (2) *soluţia prevenirii pierderii mesajelor*. In acest caz se informează procesele care comunică cu procesul ce urmează a migra despre această intentie. Pe timpul suspendării procesului pentru migrare, mesajele se buferează; în urma terminării migrării procesului se transmit mesaje de tip *sfarsit migrare* şi următoarele se transmit direct la noua adresă; (3) *soluţia recuperării mesajelor pierdute*. In acest caz, nu se salvează mesajele transmise în timpul migrării (ele sunt astfel, pierdute); totusi se memorează numele şi adresele proceselor emiţătoare, urmînd ca după migrare procesul respectiv să restabilească legatura cu aceste procese în vederea recuperării mesajelor.

BUPT

Mecanismul de implementare al migrării poate fi impartit în două faze: *faza de colectare de informații statistice* și *faza de transfer a procesului.* Colectarea de informații statistice, în general relative la încărcarea procesorului (exprimată în numarul de procese gata de execuție), a liniilor de comunicație (exprimate în numarul de mesaje în asteptare) se poate face fie periodic, fie la aparitia unor evenimente precum terminarea proceselor sau recepția mesajelor. Faza de transfer a procesului prezintă particularități legate de sistem. Se prezintă pentru exemplificare, etapele necesare pentru transferul unui proces în sistemul DEMOS/MP, creat la Universitatea din California, Berkeley . In DEMOS/MP transferul unui proces, care are loc după un proces de negociere, în urma căruia se stabileste nodul destinatar, se desfășoară astfel: (1) se marchează procesul *în migrare*, starea procesului se retine (cu exceptia stării *ready*, cînd procesul este scos din lista *ready*) pentru că procesul își va mentine starea în urma migrării; mesajele în tranzit în timpul migrării vor fi plasate în coada de mesaje a procesorului; (2) deoarece în DEMOS/MP, nucleul nodului destinatar este cel care realizează transferul, se trimite de către nodul sursă un mesaj care contine informațiile necesare pentru migrare: dimensiunea și localizarea contextului procesului, informații relative la swapping; (3) se alocă un nou context al procesului la nodul destinatar; (4) se transferă contextul; (5) se transferă codul procesului; (6) se transferă toate mesajele în tranzit din timpul migrării; totodată nodul sursă modifică adresa procesului astfel încît să reflecte noua localizare a procesului; (7) se înlatură identificatorul procesului din toate tabelele nodului sursa; (8) se restartează procesul în nodul destinație.

Sistemul Emerald exemplifică un alt aspect al migrării: acela al unității de migrare. In Emerald ([Jul88]) unitatea de distribuție și mobilitate este *obiectul.* Unele obiecte conțin procese, altele conțin numai date. Principalul scop al sistemului Emerald a fost de a experimenta mobilitatea în programarea distribuită, subiect al mobilității fiind atît procesele cît și datele.

Un alt aspect important în gestionarea procesoarelor este ***alocarea proceselor la procesoare***. Există două soluții: în prima soluție, programatorul asociază procesele la procesoare în mod explicit; în acest caz sistemul de operare este simplu, dar programatorul trebuie să fie implicat în toate deciziile de alocare. In cea de a doua soluție, distribuirea configurației programului este controlată de sistem: sistemul repartizează procesele pe procesoare și poate provoca migrarea acestor procese. Soluția prezintă avantajul unei transparente complete relativ la utilizator, dar și dezavantajul unui sistem de operare mai sofisticat. O strategie de echilibrare a încărcării procesoarelor are două aspecte: *strategia de partajare* a procesoarelor care asigură încărcarea tuturor procesoarelor, ocolind situația în care unele procesoare sunt libere și altele supraincărcate și *strategia de echilibrare a încărcării* care asigură egalarea sarcinii de lucru pentru toate procesoarele. Ambele strategii utilizează migrarea proceselor și pot fi divizate în trei: (1) *strategia de transfer*, care determină cînd trebuie executat un proces local sau la distanță; (2) *strategia de selectie,* care determină care proces trebuie transferat; (3) *strategia de localizare,* care determină pe care nod treb uie selectat procesul care trebuie transferat.

Sistemele care folosesc partajarea procesoarelor pot utiliza o gamă largă de algoritmi pentru planificării încărcării începînd cu variante complet centralizate și terminînd cu variante complet distribuite.

In *varianta centralizata,* sistemul colectează informații despre localizarea nodurilor fără sarcină de lucru, procesele care asteaptă pentru execuția la distanță și de cît timp, parametrii proceselor care se execută și unde se execută si, în functie de aceste informații, decide ce proces va primi un anumit procesor. Soluția centralizată are însă două dezavantaje: primul este legat de vulnerabilitatea la defectiunile hardware iar al doilea de acuratetea informațiilor colectate, greu de mentinut datorită intîrzierilor introduse de comunicații.

In ceea ce privește *variantele distribuite*, se enumară patru algoritmi cunoscuti și relativ usor de implementat: (1) algoritmul care organizează procesoarele într-o ierarhie logică ([Wit80], [Gos91], [Tan93], [Tan96]); (2) algoritmul care organizează procesoarele într-un inel logic [Gos91]; (3) algoritmul programelor de tip *worm*; (4) algoritmul *planificării Condor*; dintre aceștia, vor fi prezentați primii doi.

Algoritmul (1) utilizează o împărțire a nodurilor într-o ierarhie logică (fig. 1.3.2.1.5.) în care unele noduri sunt servanți, accesibili global, iar altele gestionează sarcină de lucru a servanților; conexiunile din figură nu reprezintă liniile de comunicație fizice, ci conexiunile logice in scopul gestionării nodurilor. Un nod manager gestionează încărcarea servanților din subarborele său. Alocarea procesoarelor, pentru un job care necesită *m* procese se realizează astfel (se presupune, pentru simplificare, că procesoarele sunt monoprogramate): (1) fiecare manager cunoaște aproximativ numărul de servanți disponibili din subarborele său; numărul exact nu este cunoscut, deoarece informațiile se transmit periodic. Pentru un job cu *m* procese vor fi necesare *m* procesoare; (2) dacă numarul de procesoare este suficient, adică $S>m$, managerul rezervă un numar de *m* servanți și creează contextul pentru *m* procese; (3) dacă numarul de procesoare nu este suficient, managerul transmite cererea, către predecesorul său din arborele de organizare logica a procesoarelor. Acesta are informații pentru un subarbore mai mare (numărul de nivele nu este limitat); dacă nici predecesorul nu poate satisface cererea, aceasta este transmisă și mai departe (la predecesorul predecesorului) s.a.m.d. Exista două probleme pe care le pune acest algoritm: (1) prima se referă la situația în care un manager are o defectiune. In acest caz, dacă in arbore există un predecesor al acestui manager, acesta comunică cu servanții și selectează unul dintre acestia pentru a indeplini in continuare funcția de manager. Dacă nu există un predecesor al managerului, atunci se foloseste faptul că radacină structurii ierarhice nu este unica; nodul duplicat a fost introdus tocmai pentru protecția impotriva defectiunilor. Nodurile de pe nivelul radacină comunică periodic, transmitîndu-si informații relative la subarborii lor; prin urmare, nodul duplicat poate selecta un alt manager pentru nivelul radacinii; (2) a doua problemă se referă la faptul că, dacă mai multi manageri alocă procesoare, atunci poate apare o interblocare; aceasta implică un mecanism de sincronizare între manageri

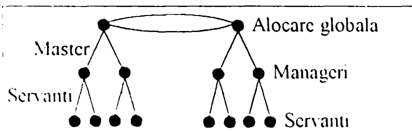


Fig. 1.3.2.1.5. O ierarhie de procesoare.

BUPT

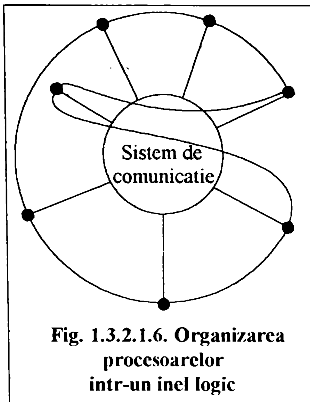


Fig. 1.3.2.1.6. Organizarea procesoarelor intr-un inel logic

Cel de al doilea algoritm organizează procesoarele într-un inel logic (figura 1.3.2.1.6.); fiecare nod poate fi şi manager şi servant. Pentru a asigura alocarea procesoarelor se transmite un token prin inelul logic. Dacă un job care necesită *m* procese apare în oricare nod, el trebuie să astepte sosirea token-ului în nodul respectiv. Alocarea procesoarelor unui job ce necesită *m* procese se desfăşoară astfel (se presupune, la fel ca în algoritmul precedent, că nodurile sunt monoprogramate): (1) deoarece nodurile nu pastrează informaţii despre disponibilitatea altor noduri din sistemul distribuit, nodul cu rol de manager (cel care trebuie să aloce procesoare pentru job) transmite un mesaj broadcast, pentru a se informa despre disponibilitatea procesoarelor. Dacă un procesor este disponibil, trimite un mesaj de confirmare şi managerul îl alocă. Dacă numarul de procesoare disponibile este mai mic decît numarul de procese, atunci managerul eliberează procesoarele alocate deja. Pe de altă parte, dacă un nod care şi-a transmis disponibilitatea nu primeşte în decursul unui interval de timp fixat o cerere de creare al unui context de proces, procesorul este eliberat. (2) se transmite token-ul următorului proces din inel; (3) dacă numarul de procesoare disponibile (servanţi) a fost gasit suficient, atunci managerul execută toate operaţiile necesare pentru crearea contextului proceselor în nodurile servanţilor; (4) procesoarele servanţi sunt eliberate după terminarea proceselor respective.

O altă variantă a algoritmului în inel este aceea în care token-ul transportă informaţii despre disponibilitatea procesoarelor în inel. La primirea token-ului, un nod manager, contorizează numarul de procesoare disponibile, *d*, şi dacă *d*<=k atunci inregistrează localizarea lor şi actualizează informaţiile din token, după care transmite token-ul mai departe. Acest algoritm (în ambele variante) ridică două probleme: pierderea token-ului şi distrugerea inelului în urma defectării unui nod; pentru acest tip de probleme, soluţiile sunt prezentate în subcapitolul 4.4.

Strategia de echilibrare a încărcării procesoarelor spre deosebire de *strategia de partajare a procesoarelor* trebuie să asigure egalarea sarcinii de lucru a procesoarelor. Strategia de partajare a procesoarelor utilizează migrarea proceselor numai cînd un procesor devine liber; în schimb, strategia de echilibrare poate utiliza migrarea chiar dacă un nod nu este liber. O clasificare a algoritmilor de echilibrare a încărcării este prezentată în fig. 1.3.2.1.7. ([Gos91]).

Algoritmii statici au drept scop determinarea procesorului optim pentru un proces în condiţiile în care procesul nu

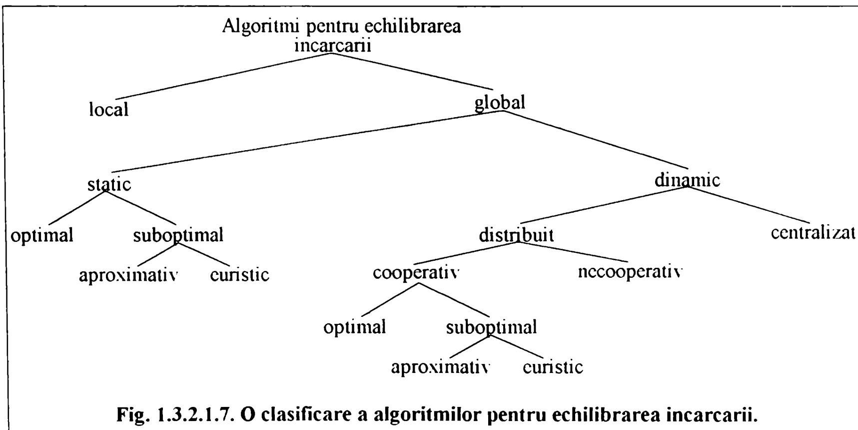


Fig. 1.3.2.1.7. O clasificare a algoritmilor pentru echilibrarea incarcarii.

va mai putea fi mutat de pe acel nod; deciziile sunt luate pe criterii probabilistice, fără a se lua în considerare starea curentă a sistemului distribuit. Echilibrarea încărcării în mod static se reduce astfel la o problemă de matematica: determinarea stării sistemului şi a cerintelor de resurse în condiţiile în care trebuie optimizaţi anumiti indici de performanţa, pe baza unor funcţii luate drept criteriu (minimizarea timpului total de execuţie al unui proces, maximizarea utilizării resurselor, maximizarea productivităţii procesorului -a numarului de procese executate în unitatea de timp). Algoritmii *statici* au fost dezvoltaţi pe baza teoriei cozilor, a metodei de programare *branch and bound şi* a teoriei grafurilor. Soluţiile *optimale* pot fi foarte costisitoare în ceea ce priveşte timpul şi de aceea se implementează în general soluţii *suboptimale*: euristice sau aproximative. Soluţiile *aproximative* nu explorează întreg spaţiul soluţiilor, scopul fiind determinarea unei soluţii satisfăcătoare. *Echilibrarea încărcării în mod dinamic* are drept scop egalarea sarcinii de lucru a procesoarelor pe masură ce procesele sunt create, suspendate şi reluate, deciziile fiind luate utilizînd informaţii relative la starea curentă a sistemului. Găsirea unei soluţii dinamice este mai complicat de realizat: este necesară colectarea de informaţii de stare, migrarea proceselor; totusi, o soluţie dinamică este mai performantă decat una statică, deoarece ţine cont de valoarea reală a indicilor pentru încărcarea sistemului.

BUPT

1.3.2.2. Gestiunea fişierelor

Diferentele majore între sistemele de gestiune ale fişierelor pentru sisteme centralizate şi cele pentru reţelele de calculatoare decurg din problemele de fiabilitate puse de ultimele deoarece trebuie prevenită apariţia inconsistentelor datorate unei avarii accidentale a unui nod (client sau server), a liniei de comunicaţii sau datorate accesului concurent la fişierele din reţea[Mus95d][Mus96f].

In contextul gestiunii distribuite a fişierelor se definesc următorii tenneni: un *serviciu* este o entitate software care se execută pe unul sau mai multe noduri şi care furnizează o anumită funcţie, un *server* este un proces care se execută pe un singur nod şi implementează un serviciu iar un *client* este un proces care poate invoca un serviciu utilizînd un set de operaţii care formează *interfaţa clientului.*

In reţelele de calculatoare, pentru gestiunea fişierelor există, în general, două soluţii:

1. ***Soluţia semidistribuită*** prin care se distribuie toate cererile de acces la fişierele din reţea către unul sau mai multe servere de fişiere (un calculator dedicat).
2. ***Soluţia complet distribuită*** în care fiecare nod mentine sistemul sau local de fişiere şi comunică cu orice alt nod al reţelei; localizarea fişierelor poate sau nu poate fi transparentă pentru utilizatori. In primul caz sistemul de gestiune mentine tabele şi foloseste algoritmi pentru a gasi un fişier oriunde în reţea. Un astfel de sistem de fişiere distribuit permite efectuarea de operaţii la distanţă asupra fişierelor, în aceeasi manieră ca şi asupra celor locale. Un exemplu este sistemul distribuit de fişiere proiectat de Sun MicroSystems, NFS (Network File System) şi implementat pe microcalculatoare pînă la supercomputere (în reţele eterogene) în care fiecare nod poate fi server si/sau client; un sistem de fişiere al unui server apare ca parte componentă a sistemului de fişiere al unui client la distanţă; scrierea şi citirea unui bloc se face prin transmiterea sa prin reţea ; deschiderea fişierelor se face la distanţă. Alte exemple sunt sistemul de gestiune fişiere al sistemului de operare distribuit LOCUS (compatibil Unix) dezvoltat la UCLA ([215],[1],[8]) în care localizarea fişierelor, transparentă pentru utilizatori, se face printr-un protocol bine stabilit iar pentru protecţie se mentin mai multe copii pentru un fişier şi Apollo Domain dezvoltat de Apollo Computer Inc.

Un *sistem distribuit de fişiere (DFS)* este o implementare distribuită a modelului clasic time-sharing de partajare a fişierelor; sistemul de fişiere al Unix-ului este deseori luat ca model.

O *clasificare* a serverelor de fişiere este prezentată în următorul tabel([Gos91]).

	Server simplu	Server universal	Suport pentru baze de date
Unitate de acces la date	*fisier*	*secventa de octeti*	*inregistrare*
Unitate de blocare	*fisier*	*fisier*	*inregistrare*
Domeniul unei actualizari atomice	*fisier*	*mai multe fisiere mai multe servere*	*mai multe fisiere mai multe servere*

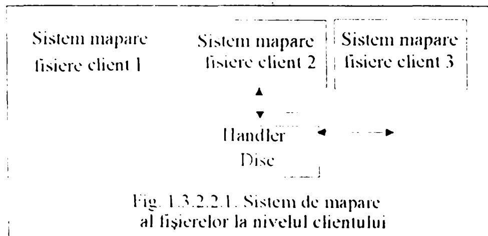


Fig. 1.3.2.2.1. Sistem de mapare al fişierelor la nivelul clientului

Cea mai simplă organizare a unui server pentru fişiere este ca un remote disc al fiecarui client. In acest model, fiecare client are alocat un disc virtual, care este de fapt o zonă din discul serverului şi pe care il adreseaza la fel ca pe un disc local. Reteaua de comunicaţii are rolul de a simula un controller de disc la distanţă, serverul furnizind numai execuţia comenzilor *read, write* la nivel de bloc la fel ca şi un driver de disc. Intregul sistem de gestiune al fişierelor la distanţă se află in nodul clientului, la fel ca şi pentru discurile locale.

Următoarea treaptă de complexitate in organizarea serverelor pentru fişiere este ca acestea să gestioneze fişiere, dar cataloagele care inventariază fişierele unui client să fie mentinute de către client (fig. 1.3.2.2.1.):

Serverul gestioneaza alocarea şi dealocarea spaţiului pe disc (dinamica) şi furnizează clientului o interfaţă pentru comunicaţie, care include operaţii (creare, stergere fişier, citire, scriere, poziţionare intr-un fişier) implementate intr-un protocol de comunicaţie. La nivelul serverului se mentin identificatori globali ai fişierelor, accesibili eventual tuturor clienţilor din reţea. La crearea fişierului se genereaza şi se returneaza clientului un identificator de fişier, FID, (in general numar de 8-64 biti); clientul poate transmite FID-ul altor clienţi pentru a avea acces la acel fişier şi insereaza FID-ul in catalogul fişierelor proprii (figura 1.3.2.2.2).

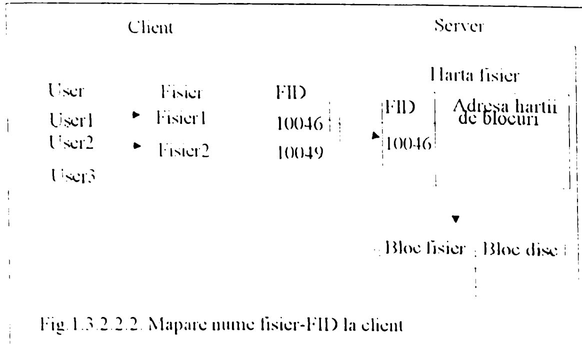


Fig.1.3.2.2.2. Mapare nume fisier-FID la client

BUPT

Pentru a evita situaţia în care ramin fişiere inaccesibile în urma esuării inregistrării FID-ului lor de către clienţi, serverul trebuie să furnizeze o comandă prin care un client să poată cere o listă completă a fişierelor sale. Se pot incorpora în FID, alături de un număr aleator şi informaţii relative la localizarea fişierelor facînd imposibilă deplasarea fişierului. Unele reţele, într-o astfel de organizare a serverului oferă clienţilor şi serviciile de catalog (*directory service*) [Tan93] (fig. 1.3.2.2.3); aceste servicii sunt cumulate într-un proces separat, rezident în nodul serverului, care apare ca un client pentru server şi ca server pentru client. Izolarea serviciilor de catalog face posibilă menţinerea mai multor scheme de evidenţă şi denumire a fişierelor utilizînd însă aceleaşi servicii de server.

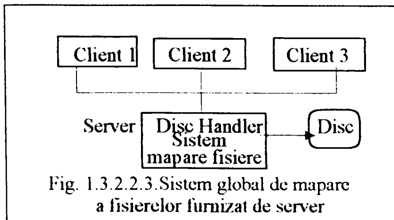


Fig. 1.3.2.2.3.Sistem global de mapare a fisierelor furnizat de server

Această structurare este echivalentă cu următoarea treaptă de organizare a unui server -aceea în care serverul furnizează un sistem de gestiune fişiere complet, posibil al unui sistem de operare cunoscut şi poate trata alături de comenzile pentru manipularea fişierelor şi comenzi pentru creare şi stergere cataloage, creare de legaturi.

In ceea ce priveşte organizarea serverului, cea mai simplă organizare este de a-l considera ca un singur proces avînd următoarea structura:

```
do true -->
    GetMessage(buffer);
    DoWork(buffer,result);
    SendReply;
od
```

Dezavantajul este ca prelucrarea unei cereri necesită în general mai multe accese la disc, timp în care serverul nu mai poate accepta noi cereri.

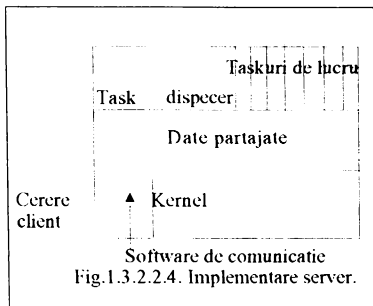


Fig.1.3.2.2.4. Implementare server.

O variantă mai complexă (fig 1.3.2.2.4) este aceea în care serverul este descompus în mai multe *task-uri de lucru* concurente care partajează structuri de date comune precum tabele ale sistemului de fişiere şi buffere, un *kernel* care acceptă mesajele de la clienţi şi un *dispecer* care administrează încărcarea *task-urilor de lucru* (fiind în acelaşi spaţiu de adrese se transmite taskurilor o referinţă la mesaje).

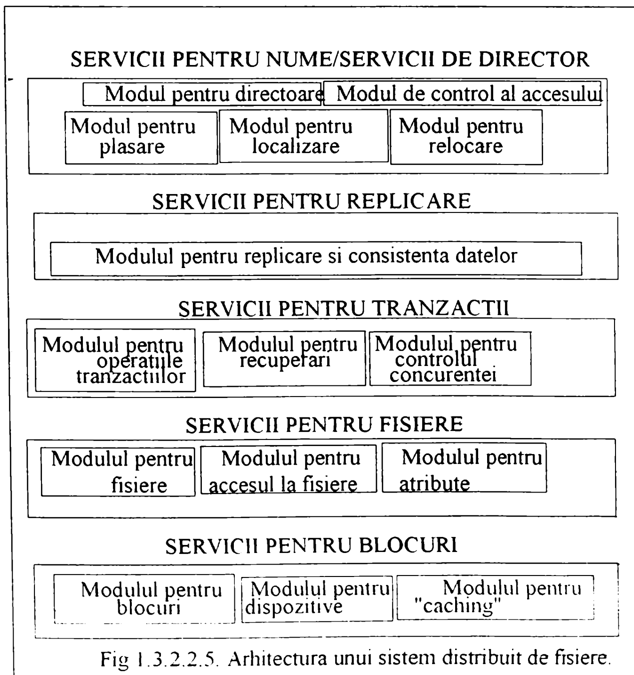


Fig 1.3.2.2.5. Arhitectura unui sistem distribuit de fisiere.

O analiză a sistemelor distribuite pentru gestiunea fişierelor din [Gos91].[Nut92].[Mul89].[Tan93] reliefează elemente comune care permit propunerea arhitecturii generale prezentate în figura 1.3.2.2.5. pentru un astfel de sistem.

Serviciile furnizate de sistemele distribuite pentru gestiunea fişierelor sunt asigurate de următoarele module:

BUPT

1. Modulul pentru dispozitive care asigură execuţia operaţiilor de I/O şi gestiunea bufferelor.
2. Modulul pentru blocuri care alocă blocurile de disc.
3. Modulul pentru implementarea mecanismului de "caching" care păstrează rezultatele obţinute recent prin operaţii cu discul sau operaţii la distanţă.
4. Modulul pentru fişiere care asociază identificatori fişierelor.
5. Modulul pentru accesul la fişiere care asigură citirea şi scrierea datelor în fişiere.
6. Modulul pentru atribute care citeste şi scrie atributele asociate fişierelor.
7. Modulul pentru operaţiile tranzacţiilor care furnizează facilitatea lucrului cu tranzacţii atomice.
8. Modulul pentru recuperari care furnizează mecanismul de recuperare după caderea unor componente.
9. Modulul pentru controlul concurenţei care asigură consistenţa datelor în fişierele partajate.
10. Modulul pentru replicare şi consistenţa datelor care asigură replicarea fişierelor şi pastrarea consistenţei copiilor.
11. Modulul pentru directoare care asigură corespondenţa între numele date de utilizatori fişierelor şi identificatorii fişierelor.
12. Modulul de control al accesului care asigură verificarea drepturilor de acces pentru operaţia ceruta.
13. Modulul pentru plasare care selectează serverul ce va primi o nouă copie a fişierului.
14. Modulul pentru localizare care găseşte serverul ce gestionează un anumit fişier.
15. Modulul pentru relocare care imbunatateste performanţele sistemului prin relocarea fişierelor.

Separarea serviciilor pentru blocurile discurilor de cele pentru fişiere este o caracteristică a majorităţii sistemelor distribuite de fişiere; această separare asigură, pe de o parte posibilitatea combinării a diferite metode de stocare si, pe de o parte, utilizarea unei stocari duale (*stable storage*).

Din punctul de vedere al utilizatorilor, serverele pentru fişiere se caracterizează prin: *structura fişierelor, atributele lor şi operaţiile permise* asupra fişierelor[Tan88][Mus96f][Mus96d].

Există trei ***modele de structuri de fişiere*** utilizate de serverele pentru fişiere: primul model este *fişierul brut*, a cărui structură internă nu este cunoscută de server; singurele operaţii posibile sunt scrierea şi citirea întregului fişier; următorul model este fişierul constituit dintr-o *secvenţă ordonată de articole*, eventual nu toate de aceasi dimensiune. Serverul poate furniza operaţii precum adăugare, înlocuire sau ştergere de articole. Un fişier Unix poate fi privit ca avînd articole de un octet, accesibile prin poziţie. Modelul cel mai complex este *fişierul ierarhic*, organizat pe o structură de arbore. Aceasta poate fi o structură de tip *secvential indexat* (VISAM - Virtual Indexed Sequential Access Method) în care fiecare articol se identifică printr-o cheie, sau o structură arborescentă mai generală în care fiecare nod al unui arbore poate conţine o etichetă, un articol, amîndouă sau nici unul. Un exemplu este prezentat în fig.1.3.2.2.6.: un arbore cu 12 noduri, două avînd etichete (a şi b) şi nouă noduri avînd articole (toate nodurile exceptînd 1, 2 şi 7). Prin liniarizarea arborelui, nodurile se ordonează (în exemplu prin parcurgere în Depth First), nu numai pentru o cautare mai rapidă a nodurilor neetichetate în scopul ştergerii sau înlocuirii lor, dar şi pentru adresarea convenabilă a subarborilor, în scopul efectuării de operaţii la nivel de subarbori.

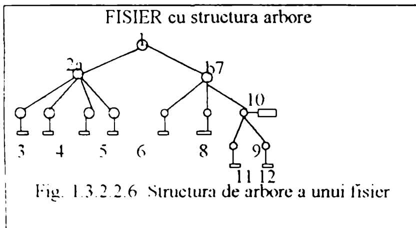


Fig. 1.3.2.2.6 Structura de arbore a unui fisier

Atributele asociate fişierelor diferă mult de la un server la altul. Unele atribute, stabilite la crearea fişierelor ramin neschimbate, altele pot fi modificate explicit prin operaţii sau actualizate automat de server.

Standardul OSI [10] prevede pentru fişiere atributele prezentate în fig. 1.3.2.2.7. precum şi operaţiile prezentate în figura 1.3.2.2.8.

Atribut	Tip	Setat la creare	Setat de user	Setat de server
Nume fisier	String	x	x	
Operatii permise	Harta biti	x		
Control accese	Lista			
Numar de cont	INT	x		
Data creerii	DATA	x		
Data ultimei modificari	DATA	x		
Data ultimei citiri	DATA	x		
Data ultimei modificari a atributelor	DATA	x		
Proprietar	UserID	x		
Identitatea ultimului modificator	UserID	x		
Identitatea ultimului cititor	UserID	x		x
Identitatea ultimului modificator al atributelor	UserID	x		
Tipul continutului	Obiect	x		
Cheie de criptare	String	x		
Dimensiune	Long	x		
Dimensiune maxima evaluata	Long	x	x	
Calificatori legali	String	x	x	
Utilizare proprie	String	x	x	

Fig. 1.3.2.2.7. Atribute asociate fisierelor

Operatie	Aplicabila fisier	Aplicabila articol	Bitmap
Creare fisier	x		
Stergere fisier	x		x
Selectie fisier	x		
Deselectie fisier	x		
Inchidere fisier	x		
Citire atribute	x		x
Modificare atribute	x		x
Localizare		x	
Citire		x	x
Inserare		x	x
Inlocuire		x	x
Adaugare		x	x

Fig 1. 3.2.2.8. Operatii asociate fisierelor.

Pentru *comunicarea între clienţi şi serverele de fişiere* se folosesc protocoale care trebuie să satisfacă două cerinţe specifice reţelelor: (1) controlul fluxului de mesaje dintre server şi client astfel încît nici unul să nu fie saturat (2) controlul erorilor datorate transmisiei prin liniile de comunicaţie, sau defectării serverului sau clientului. Astfel, unele servere impun clienţilor lor să emită cel puţin o cerere într-un interval de timp maxim fixat. In absenţa unor astfel de mesaje de tip *keep alive*, serverul abortează tranzacţia asociată clientului. Trebuie luate în considerare două aspecte: clienţi care nu răspund şi cereri duplicate (răspunsul serverului nu a fost recepţionat de client care lansează o nouă cerere).

Se utilizează în general următoarele *protocoale*: (1) *protocolul în doi paşi*, bazat pe modelul *send - request / receive - response* (LOCUS, DEMOS); (2*) protocolul în trei paşi*, bazat pe modelul *send - request / receive - response / send - acknowledgement* (RRA) (SWALLOW;) (3) *protocolul într-un singur pas, no-wait send, (single shot*) care impune ca toate cererile să fie *idempotente* (execuţia repetată are acelaşi efect ca şi o singură execuţie);(4) *apel de procedură la distanţă* (NFS). Parametrii fiecarui apel de procedură executate în cadrul protocolului NFS, bazat pe RPC, conţine toate informaţiile necesare execuţiei unei cereri, sau, altfel spus, serverul nu reţine nici o informaţie din cadrul cererii precedente; astfel de servere se numesc *servere care nu pastrează informaţii de stare* (stateless servers) şi prezintă avantajul unei recuperari simple după o avarie a serverului (se repetă cererea clientului). Pe de altă parte, însă, un server care nu pastrează informaţii de stare are şi următoarele dezavantaje: (1) este un model inadecvat pentru tratarea acceselor concurente a mai multor clienţi; (2) cererile sunt mai lungi, avînd incorporaţi mulţi parametrii.

Mecanismul utilizat pentru a retrage resursele alocate de server unui client care nu mai răspunde sunt: (1) *timeout* la nivelul unei tranzacţii (SWALLOW); (2) *timeout* la nivelul cererilor; (3) transmiterea de mesaje *keep-alive*.

Un alt aspect important al gestiunii distribuite a fişierelor este ***transparenţa***, ceea ce implică faptul că clienţii pot accesa fişiere la distanţă utilizînd aceleaşi operaţii aplicabile fişierelor locale. Aceasta înseamnă că interfaţa clienţilor într-un sistem distribuit este uniformă, nu distinge fişierele locale de cele la distanţă; este sarcina sistemului să localizeze fişierele şi să realizeze transportul datelor. Legată de transparenţă este problema numelor fişierelor. Regula de formare a numelor fişierelor stabileste o corespondenţă între obiectele logice cu care operează utilizatorii şi obiectele fizice (blocurile fizice ale fişierului); ea trebuie să asigure, pe de o parte *transparenţa localizării* (numele unui fişier nu trebuie să contină indicatii relative la locatia sa fizică) şi pe de altă parte *independenţa faţă de localizare* (numele unui fişier nu trebuie să se modifice cînd fişierul este mutat).

Independenţa faţă de localizare, adeseori numită şi *migrarea fişierelor* sau *mobilitatea fişierelor* este o proprietate mai puţin întîlnită la sistemele distribuite, care oferă în general o mapare statica, şi transparenţa localizării (Locus, NFS, Sprite). Independenţa faţă de localizare este oferită de Andrews şi cîteva sisteme distribuite experimentale (Eden, de exemplu).

Există trei tehnici pentru denumirea directoarelor şi fişierelor:

(1) cea mai simplă tehnică este prin combinarea numelui nodului şi fişierelor, care garantează astfel unicitatea în tot sistemul: *host: nume_local*. Evident că tehnica nu oferă nici transparenţa localizării şi nici independenţa faţă de

BUPT

localizare; totusi, pot fi folosite aceleaşi operaţii atît pentru fişierele la distanţă, cît şi pentru cele de la distanţă. Structura DFS-ului constă într-o colectie de componente izolate, care impreună formează un sistem de fişiere.
(2) a doua tehnică, popularizată de SUN NFS se bazează pe operaţia de *montare* a directoarelor de la distanţă într-un spaţiu de nume local; după această ataşare, fişierele pot fi denumite într-o manieră transparentă din punct de vedere al localizării. O proprietate a acestei tehnici, uneori considerată un dezavantaj, este aceea că spatiile de nume paratajate în sistem nu sunt identice pentru toate nodurile;
(3) a treia tehnică oferă o structură globală de nume în sistem, care cuprinde toate nodurile; întregul spaţiu de nume este vizibil tuturor clienţilor. In ceea ce priveşte implementarea acestei tehnici, se specifică următoarele aspecte: (a) *translatarea cailor*, în fig. 1.3.2.2.9.a si1.3.2.2.9.b se prezintă două modalităţi de determinare a localizării unui fişier /a/b/c; în fig. 1.3.2.2.9.a cautarea se realizează la initiativa clientului (varianta adoptată în Andrew şi Locus), iar în fig. 1.3.2.2.9.b căutarea se realizează la iniţiativa serverului (variantă adoptată în NFS şi Sprite); (b) *identificatorii fişierelor*, care reprezintă numele fişierelor la nivel de sistem , uzual întregi de lungime fixa, se pot obţine prin concatenarea următoarelor campuri: adresa serverului carea a creat fişierul, o marcă de timp de la ceasul serverului sau un întreg care reprezintă valoarea unui contor al serverului incrementat la crearea fiecarui fişier. Asemenea identificatori prezintă avantajul că identifică în mod unic un singur obiect fişier în întreg fişierul, oferă independenţă de localizare (adresa serverului este necesară numai pentru asigurarea unicităţii, nu şi pentru localizare), pot fi transmisi între procese şi de la nod la nod, fără a fi nevoie de transformarea lor la fiecare pas. (c) folosirea unei proceduri de *cache*-ing pentru informaţiile de localizare la nivelul clienţilor. Exemple sunt sistemul Andrews, sistemul Apollo Domain şi sistemul Sprite care utilizează un mecanism similar *cache*-ing-ului, aşa numitele *tabele de prefix,* iar cînd serverul cautat nu se află în aceste tabele, transmite mesaje broadcast,.

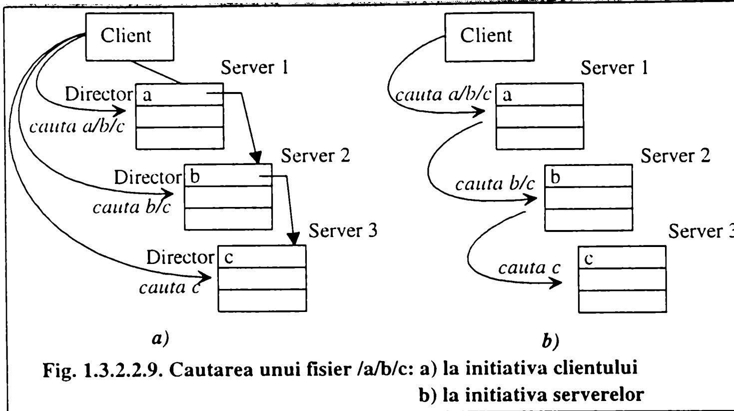


Fig. 1.3.2.2.9. Cautarea unui fisier /a/b/c: a) la initiativa clientului b) la initiativa serverelor

Un alt aspect important într-un DFS este ***partajarea fişierelor***. In acest sens, există următoarele implementari; (1) utlizarea *modelului semantic Unix*; în acest caz toate cererile de citire şi scriere se adresează direct serverului, care le procesează secvential. Unele DFS-uri, precum Locus, Sprite, încearcă să emuleze această semantica. Soluţia este aplicabilă în practică în special în cazul unui singur server şi în lipsa cache-ing-ului la nivelul clienţilor; ; (2) o alternativă la semantică Unix este întîrzierea propagării modificarilor în fişier, pînă în momentul închiderii fişierelor (*semantica de sesiune*). Problema pe care o ridică acest model este situaţia în care doi sau mai multi clienţi modifică independent o copie cache a fişierului. O soluţie este ca, la închiderea fişierului, acesta să fie înapoiat serverului. Acest model este însă adecvat pentru aplicaţiile la care serializarea acceselor este esentială (baze de date); aceste aplicaţii trebuie să-şi coordoneze accesul explicit. (3) O implementare diferită de cele de mai sus este cea a *fişierelor partajabile imuabile*; acestea, după creare, nu mai pot fi alterate, iar numele lor nu mai pot fi reutilizate. In acest caz, fiecare fişier este reprezentat printr-o istorie de fişiere imuabile; se crează cîte o nouă versiune, la fiecare încercare de modificare; (4) o extindere a modelului *semantică de sesiune* este *modelul tranzacţiilor atomice*: efectul sesiunilor este echivalent cu execuţia lor seriala, într-o anumită ordine. Blocarea fişierului pe parcursul unei sesiuni este o soluţie pentru realizarea acestui model.

Un alt aspect al unui DFS este implementarea ***accesului la distanţă***; în acest sens există următoarele soluţii:

(1) rezolvarea cererilor clienţilor numai la nivelul serverului (servicii la distanţă): serverul recepţionează şi tratează cererile şi transmite rezultatele înapoi la client. In această implementare există o corespondenţă directă între cererile clienţilor şi traficul mesajelor;

(2) utilizarea unui mecanism de *cache*-ing (la nivelul serviciului si/sau nivelul clienţilor). Există următoarele decizii importante relative la proiectarea unui mecanism de *cache*-ing: (a) *granularitatea datelor* mentinute în *cache*: întregul fişier sau blocuri din fişier. De exemplu, sistemul Andrews, după o versiune în care mentinea în *cache* întreg fişierul, a fost rescris într-o versiune în care in *cache* se mentin cate 64Kb din fişier. Prin creşterea unităţii de memorare în *cache*, creşte evident probabilitatea ca următorul acces să fie realizat local, dar creşte şi probabilitatea aparitiei problemelor de consistenţă. Pe de altă parte, deoarece unitatea de transfer în reţele este relativ mică (pachetele Ethernet au aproximativ 1,5Kb) o unitate de memorare în *cache* mare necesită operaţii de impachetare la emisie şi despachetare la recepţie. In mod tipic *cache*-ingul la nivel de bloc utilizează tehnica *read-ahead* pentru citirea fişierelor. (b) *localizarea cache-ului*: în memorie sau pe un disc local. Localizarea pe disc are avantajul supravietuirii datelor în urma unei defectiuni; pe de altă parte, însă localizarea în memorie permite lucrul cu statii fără discuri, iar datele pot fi accesate mai rapid. (c) *propagarea modificarilor din cache*. Există mai multe soluţii: cea mai simplă este scrierea imediată a datelor pe server, la orice modificare (strategia *write-through*) avantajul fiind siguranţa. Această strategie este însă echivalentă cu realizarea serviciilor de scriere la distanţă; numai citirea beneficiază de fapt de *cache*-ing. Altă soluţie este scrierea cu întîrziere a datelor pe server (strategia *delayed-write*)

BUPT

delayed-write)care are două variante: scrierea pe server se face periodic (de exemplu în Sprite, la fiecare 30s), fie la închiderea fişierului - *write-on-close*; (d) *validarea datelor* din *cache*. Utilizarea *cache*-ingului la nivelul clienţilor introduce problema consistenţei datelor. Există două soluţii de bază pentru verificarea validităţii datelor din *cache*: verificarea la initiativa clientului şi verificarea la initiativa serverului. In cazul primei soluţii, frecvenţa verificării este crucială: ea poate fi facută la fiecare acces la un fişier sau numai la deschiderea fişierului sau poate fi facută periodic. In cazul celei de-a doua soluţii, serverele înregistrează ce parte a fişierelor se află în *cache*-ul fiecarui client. La detectarea unui conflict (cînd cel puţin un client accesează fişierul pentru scriere), dacă de exemplu se utilizează semantica de sesiune, la închiderea oricarui fişier modificat, serverul anuntă ceilalti clienţi să-şi descarce *cache*-ul ca invalid. Dacă se implementează însă o semantică mai restrictivă (de exemplu semantica Unix) atunci serverul trebuie să înregistreze deschiderea oricarui fişier şi modul de lucru cu acesta (read sau write); serverul trebuie să interzică deschiderea fişierelor într-un mod conflictual, pur şi simplu dezactivînd *cache*-ul pentru acel fişier şi furnizînd execuţia serviciilor la distanţă (ca de exemplu în Sprite).

Multe implementari au fost realizate prin combinarea celor două soluţii de mai sus (*soluţii hibride*). Astfel, în LOCUS şi NFS implementarea se bazează pe soluţia 1 dar este completată, pentru marirea performanţelor cu un mecanism de *cache*-ing. Pe de altă parte, implementarea accesului la distanţă în Sprite se bazează pe *cache*-ing, dar, aşa cum s-a aratat, în anumite situaţii se poate adopta şi soluţia serviciului la distanţă.

Pentru creşterea performanţelor unui DFS, se foloseste deseori ***replicarea fişierelor***. Sunt trei mari avantaje oferite de replicare: (1) creşterea sigurantei de funcţionare, avînd mai multe copii ale unui fişier; (2) disponibilitatea unui fişier şi în cazul avariei unui server; (3) divizarea cererilor de acces pe mai multe servere, evitîndu-se "sufocarea" unui singur server. Principalul aspect legat de replicare este *transparenţa*. In acest sens există trei implementari: (1) controlul copiilor se face de către client; (2) la crearea fişierului, se realizează o singură copie, urmînd ca celelalte să fie realizate ulterior (verificîndu-se dacă nu cumva fişierul s-a modificat între timp (fig.1.3.2.2.10); (3) se creează toate copiile la crearea fişierului utilizîndu-se comunicaţiile de grup (fig.1.3.2.2.11).

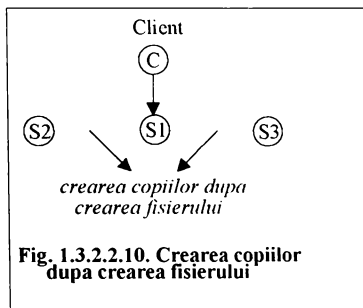


Fig. 1.3.2.2.10. Crearea copiilor dupa crearea fisierului

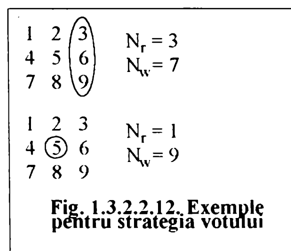


Fig. 1.3.2.2.12. Exemple pentru strategia votului

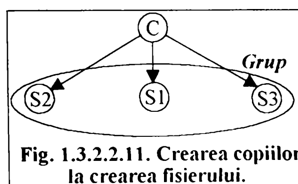


Fig. 1.3.2.2.11. Crearea copiilor la crearea fisierului.

În ceea ce priveşte modificarea copiilor fişierelor există următoarele soluţii: (1) *strategia master-slave* sau a *copiei primare* în care, pentru fiecare fişier replicat se desemnează un *server master* care contine singura copie ce poate fi actualizata. Strategia este adecvată în aplicaţii în care fişierele nu se actualizează frecvent şi în care actualizările pot fi realizate numai într-un punct central. Un exemplu de aplicare al acestei strategii este serviciul Sun Yellow Pages; (2) *strategia votului* în care ideea de bază este de a obliga clienţii să obţină permisiunea de la mai multe servere pentru a scrie sau citi un fişier replicat ([Tan88][Giff79]). Astfel, Gifford propune ca pentru ambele operaţii să existe cate un *quorum (read quorum şi write quorum)* N_r, respectiv N_w pe care clienţii trebuie să-l obţină pentru a executa operaţiile respective. Dupa obţinerea acestor permisiuni, fişierul este modificat şi i se asociază o nouă veriune. In fig.1.3.2.2.12 se prezintă două exemple.

BUPT

1.4. Modele în proiectarea sistemelor de operare distribuite

Există două modele de proiectare a unui sistem de operare distribuit: *modelul proces* și *modelul obiect.* Diferentele majore între cele două modele de proiectare constau în modul în care sunt implementate conceptul de entitate funcțională și sincronizarea.

1.4.1. Modelul proces

Modelul proces propune sistemul de operare distribuit ca fiind un ansamblu de procese, utilizate pentru controlul resurselor din rețea (fig. 1.4.1). Fiecărei resurse din rețea i se asociază un proces pentru a o gestiona. Un proces este vazut ca o zonă de cod și de date, și se află într-o anume stare.

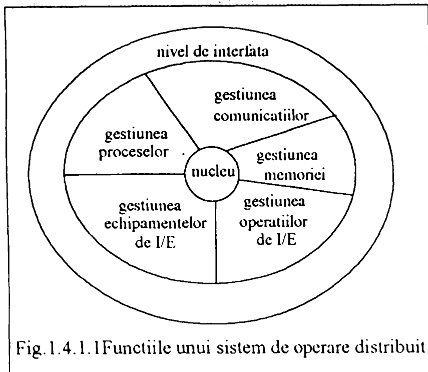


Fig. 1.4.1.1Functiile unui sistem de operare distribuit

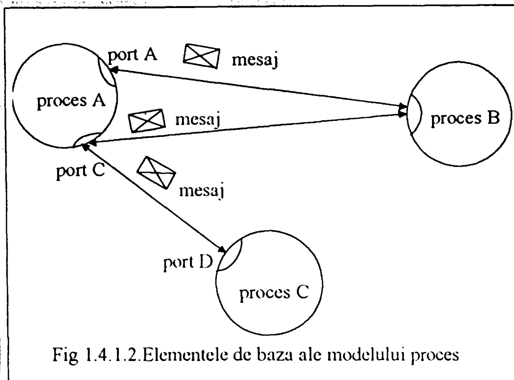


Fig 1.4.1.2.Elementele de baza ale modelului proces

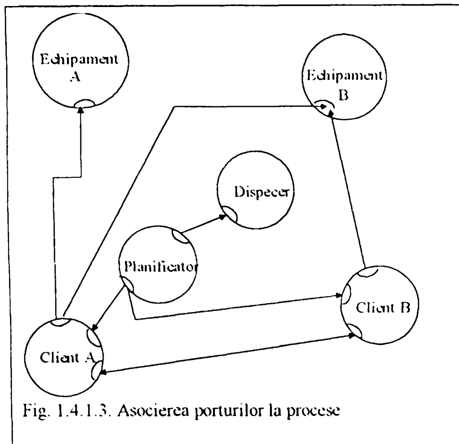


Fig. 1.4.1.3. Asocierea porturilor la procese

Procesele sistemului de operare gestionează resursele și interacționează; ele sunt utilizate de procesele aplicațiilor rezultînd astfel un sistem operațional. Procesele sistem sunt denumite *procese publice* sau *servere* (în contrast cu procesele lor utilizator denumite *clienți*). Procesele server sunt de două tipuri: *utilizabile serial* sau *paralel,* în funcție de tipul serviciului implementat. Procesele client și server comunică prin mesaje pentru a-si putea transmite informații de control, de stare și de date. Toate acțiunile din sistem sunt îndeplinite via mesaje. Perechea *(proces,mesaj)* este conceptul de bază în acest model, entitatea care îl distinge de modelul obiect: procesul devine mecanismul care implementează diverse politici de gestiune și control iar mesajele asigură cooperarea și sincronizarea interprocese (fig. 1.4.1.2).

Principalele mecanisme folosite în modelul proces pentru interconectarea logica/fizică a proceselor sunt *porturile* și *canalele.* Conceptul de *port* definește o entitate logica. Realizarea sa fizică poate consta într-un buffer adresabil, încărcat prin hardware la detectia unui identificator particular. Porturile pot fi de natură *statică* sau *dinamica.* Porturile a caror alocare se face static sunt adecvate cind o pereche de procese comunică prin porturile sursa-destinație pe tot parcursul execuției lor; este cazul situațiilor cind procesele sunt create asemanator corutinelor sau cînd procesele distribuite necesită un nivel ridicat de sincronizare (vezi subcap. 3.2.6.) In contrast, porturile dinamice sunt create, asociate, utilizate și conectate de o manieră dinamică în timpul execuției proceselor. Un exemplu tipic este utilizarea dinamică a porturilor alocate unui proces serv care controlează un echipament specializat. Procesul este accesibil pentru comunicare oricarui alt proces din sistem care necesită serviciile sale și care posedă drepturi de utilizare a echipamentului. Portul alocat procesului server este ținut permanent activ; în momentul în care un proces client doreste să utilizeze echipamentul, își crează un port (sau utilizează un port deja creat) pe care il conectează la portul procesului server, stabilește parametrii de control și utilizează ap echipamentul prin transfer de mesaje via cele două porturi. La încheierea utilizării, echipamentul periferic este eliberat ș conexiunea dintre cele două porturi este rupta. Prin conectarea proceselor de o manieră corectă se pot implementa diver sincronizari între procese precum și o multitudine de schimburi de date care pot furniza, în final, importante servicii de cooperare în rețea. De exemplu, pentru a-si aloca echipamente de intrare-ieșire și de unități de prelucrare, două procc client A și B trebuie să creeze porturile și conexiunile din fig. 1.4.1.3.

BUPT

Transferul de mesaje între procese asigură nu numai schimbul de informaţii ci şi sincronizarea între procese. Mesajele pot fi transmise de la un proces A la un proces B printr-un mecanism de cutie poştală sau utilizînd primitive pentru emisie-recepţie de mesaje. Mesajele pot fi transmise cu blocare, cind emiţătorul sau receptorul asteaptă confirmarea efectuării operaţiei de emisie, respectiv recepţie, ceea ce asigură şi sincronizarea între procese, sau fără blocare. Pentru a face mecanismul flexibil, se prevede, în general şi o variantă de transmisie fără blocare, alaturi de operaţiile implementate cu blocare (vezi subcap. 3.2.). Mecanismele fără blocare, ca de exemplu, cutiile poştale nu asigură sincronizarea implicită. Mesajele sunt transmise dintr-o cutie poştală în cealalta, receptorului i se semnalează că s-a primit un mesaj, după care, acesta extrage mesajul şi îşi continuă prelucrarea. Acest mecanism se utilizează în cazul proceselor slab cuplate, care achizitionează date unul de la celalalt în mod ocazional. Mesajele reprezintă astfel mecanismul prin care procesele îşi comunică unele altora informaţii de control, de date, de stare; schema care implementează transmisia trebuie să asigure siguranta în funcţionare. Ea trebuie să semnaleze proceselor cind s-a transmis un mesaj, cind s-a recepţionat, şi cind s-a pierdut un mesaj. Aceasta implică posibilitatea ca nivelurile inferioare ale reţelei să poată detecta şi corecta un mesaj incorect, eventual prin retransmitere.

In general, la proiectarea unui sistem de operare distribuit se are în vedere o anumită politică în conformitate cu care se tratează sincronizarea proceselor. Această politică încadrează mecanismul de sincronizare într-una din următoarele trei clase: *sincronizare slaba, sincronizare puternică sau o combinatie a acestora.* O sincronizare puternică se referă la situaţia unui proces care nu îşi poate continua prelucrarea pînă cind un alt anumit proces nu atinge punctul de sincronizare. Este cazul corutinelor. O altă formă de sincronizare puternică o constituie folosirea semafoarelor. Semafoarele furnizează proceselor garantia că se pot "intilni" într-un anumit punct dacă acest lucru este necesar. Pentru asigurarea unei sincronizari slabe, o schemă folosind cutii poştale este suficientă. In cadrul acesteia, se rezervă şi asociază o cutie poştală fiecarui proces sau unui grup de procese. Cind un proces intentionează să comunice cu un altul, trimite mesajul în cutia poştală asociată destinatarului; concomitent se setează şi un indicator care semnalează că procesul destinatar a primit un mesaj. Cînd este posibil, procesul destinatar scanează cutia poştală şi extrage mesajul respectiv.

1.4.2. Modelul obiect în proiectarea sistemelor de operare distribuite

Un sistem de operare proiectat după modelul *obiect* are o funcţionalitate similară cu modelul proces, dar conceptele utilizate pentru constructia sa sunt diferite. Modelul obiect prevede o structură de bază denumită obiect compusă din două parti: o specificatie externă şi un corp al obiectului. *Corpul obiectului* este cunoscut şi accesibil numai obiectului însuşi şi reprezintă sfera sa de control. *Specificatia externă* contine structurile sale accesibile din exterior. Conceptul de obiect semnifică atît entităţi hardware cit şi software şi este o generalizare a tipurilor de date abstracte; un obiect se caracterizează, în ceea ce priveşte reprezentarea sa, prin structurile sale de date (fişiere, directoare, liste, cutii poştale, resurse hardware, module executabile) şi prin setul invariant de operaţii ce se pot executa asupra acestor structuri de date. Obiectele pot fi şi încuibate unele în altele.

In modelul obiect este posibil să coexiste diferite tipuri de comunicaţii pentru comunicaţiile interobiect şi intraobiect. Controlul şi cooperarea între obiecte se realizează prin achizitia de capabilităţi. Astfel, accesul la o resursă în sistem se face prin achizitia unei capabilităţi de la managerul resursei respective; la terminarea utilizării, capabilitatea este eliberată şi poate fi pusă la dispozitia altor cereri.

Interacţiunea în întreg sistemul se bazează în întregime pe operaţii executate asupra obiectelor. Tipic, obiectele implementează un numar mic de operaţii, dar prin combinarea în diverse structuri a obiectelor se pot obţine funcţii complexe. Se obţine astfel o flexibilitate şi modularitate sporită într-un astfel de sistem.

Această implementare conduce la structurarea sistemului ca o colectie de resurse şi interfeţe la resurse. Sistemul apare ca o colectie de obiecte mai mult sau mai puţin dependente unele de altele. Se spune că sistemul de operare creat cu un astfel de model capată o structură orizontală [Jon78]. Sistemul apare însă şi ca o ierarhie de abstracţiuni. Aici intervine conceptul de mostenire din limbajele de programare orientate pe obiecte si, mai general, conceptul de modularitate, care conduc şi la structurarea pe verticală a sistemului. Cele mai complexe obiecte apar utilizatorului ca o interfaţă disponibilă în programare, pe baza căreia acesta poate să-şi construiască aplicaţii. Avantajul acestei implementari este că această interfaţă este, la rindul ei prevazută cu un alt set de interfeţe (posibil ierarhic) pentru accesul la obiecte. Utilizatorul poate continua o astfel de dezvoltare ierarhică şi scopul extensibilităţii modelului este atins. Modelul obiect prezintă avantajul unui protocol uniform pentru accesul la obiecte, care interacţionează, în general prin transferul de mesaje.

Luind în considerare implementarea paralelismului în modelul orientat pe obiecte, o clasificare [DG89][Mus93d] a sistemelor de operare distribuite este reprezentată în fig. 1.4.3.1.

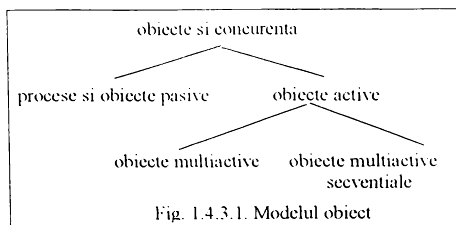


Fig. 1.4.3.1. Modelul obiect

Modelul *procese şi obiecte pasive* semnifică de fapt prezenţa simultană a două modele: proces şi obiect. Entităţile active sunt reprezentate ca procese, iar cele pasive ca obiecte. Avantajul reprezentării entităţilor pasive ca obiecte rezidă în stucturarea datelor. Exemple sunt sistemele Emerald [CC91] [DG89]. Totusi, acest model este considerat a fi la graniţa dintre modelele proces şi obiect [DG89] datorită următoarelor motive: (1). Modelul trebuie descris folosind şi entitatea de proces, pierzîndu-si

BUPT

astfel caracteristica de a fi structurat pe baza unei singure entităţi (obiectul); (2). Există două forme de interacţiune între componentele sistemului: transferul de mesaje şi cooperarea între procese. Ultima formă se reprezintă în mecanisme traditionale partajării unei memorii comune (semafoare, monitoare). Cooperarea între procese se specifică independent de prezenţa obiectelor.

Modelul *obiecte active* pastrează cele două caracteristici esentiale ale modelului obiect: încapsularea şi abstractizarea. Un obiect poate fi şi o entitate activa; el capată proprietăţile unui proces sau unui grup de procese. Aceste obiecte active se execută în paralel. Interacţiunea între obiectele active se descrie uniform printr-un singur mecanism: transfer de mesaj. La nivelul unui obiect, recepţia mesajelor va fi implementată printr-un *controller* (uneori denumit *gardian*). Sincronizarea este integrată în mecanismul de transfer. Modelul *obiecte active* poate fi în continuare divizat în două submodele: *obiecte multiactive şi obiecte active secventiale.*

In submodelul *obiecte multiactive* un obiect activ apare ca un set de activităţi, executate concurent şi corespunzînd mesajelor recepţionate de către obiect. O activitate este creată şi initiată dinamic la recepţia unui mesaj şi distrusă după încheierea tratării mesajului.Toate activităţile concurente ce se execută în cadrul unui obiect partajează o zonă de date comune, interne obiectului. Gestiunea acestor activităţi (procese) este complet disociată de gestiunea obiectelor. Submodelul este caracteristic sistemelor de operare distribuite Argus, Eden şi Mach [BF85], [DG89]. O consecinţă a acestei implementari este necesitatea sincronizării activităţilor interne, care partajează acelaşi context de execuţie. Există, de aceea, în acest model, două nivele de cooperare: între obiecte, pe baza transferului de mesaje şi între activităţile interne, pe baza partajării unei memorii comune. Astfel, se folosesc semafoare în Argus şi monitoare în Eden. Soluţia implementată în submodelul *obiecte multiactive* oferă avantajul paralelismului la nivelul obiectelor precum şi în interiorul obiectelor dar nu prezintă avantajul unui mecanism uniform de proiectare. Prezenţa a două tipuri de cooperare în sistem creşte complexitatea şi face dificilă dezvoltarea taskurilor de aplicaţie.

Submodelul *obiecte active secventiale* prevede un obiect activ ca un singur proces secvential, care tratează mesajele recepţionate. După recepţie, mesajele sunt înlănţuite în interiorul obiectului şi tratate secvential, în conformitate cu o anumită strategie (în ordinea datei de sosire-deci FIFO, în ordinea datei de emisie sau folosind marca de timp propriu).

BUPT

Capitolul 2.

COMUNICAŢII IN SISTEME DISTRIBUITE. PROTOCOALE.

Acest capitol prezintă probleme specifice ale comunicaţiei în reţele de calculatoare, ale implementării software-ului de comunicaţie precum şi aspecte specifice ale protocoalelor de comunicaţie în sisteme distribuite.

2.1. Conceptul de arhitectura deschisă. Stratificarea ierarhică a protocoalelor.

Desi modul de conectare fizică al calculatoarelor care comunică într-o reţea la mediul de transmisie ce constituie suportul reţelei poate diferi considerabil de la un tip de reţea la altul, se poate aprecia ca schimbul de informaţie între calculatoare în reţele poate fi reprezentat la modul general ca în figura 2.2.1

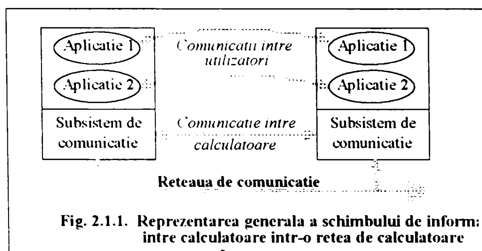


Fig. 2.1.1. Reprezentarea generala a schimbului de inform: intre calculatoare intr-o retea de calculatoare

La baza întregului schimb de informaţie poate fi o reţea locală LAN, un numar de astfel de reţele interconectate sau o reţea WAN. Totusi, indiferent de tipul reţelei de comunicaţie a datelor, fiecărui calculator trebuie să i se ataşeze un software de comunicaţie, denumit *sistem de comunicaţie*, pentru a controla stabilirea unor canale de comunicaţie între calculatoare şi pentru a controla fluxul de mesaje în cadrul acestor canale. Implementarea şi structurarea sistemului de comunicaţie este dependentă de tipul reţelei de comunicaţie. Pe de altă parte, acest software trebuie integrat într-un sistem de operare cu API-uri specifice şi apelat din limbaje de programare (cu caracteristici diferite în ceea ce priveste reprezentarea datelor); ca atare sistemul de comunicaţie este dependent şi de sistemul de operare folosit.

Proiectarea şi implementarea sistemului de comunicaţie ridică probleme mult mai mari decit proiectarea sistemului de I/O al calculatoarelor într-un sistem de operare, datorate în special fiabilităţii mai reduse a mediului de comunicaţie şi partajarea lui de către toate echipamentele legate în reţea. Dacă detaliile hardware pot fi separate într-un modul software driver, analog sistemelor de I/O, şi parţial, în controller-ul de reţea (fig 2.1.2), trebuie însă rezolvate şi alte probleme specifice precum *asigurarea unui serviciu de livrare a mesajelor sigur* şi *dirijarea mesajelor.*

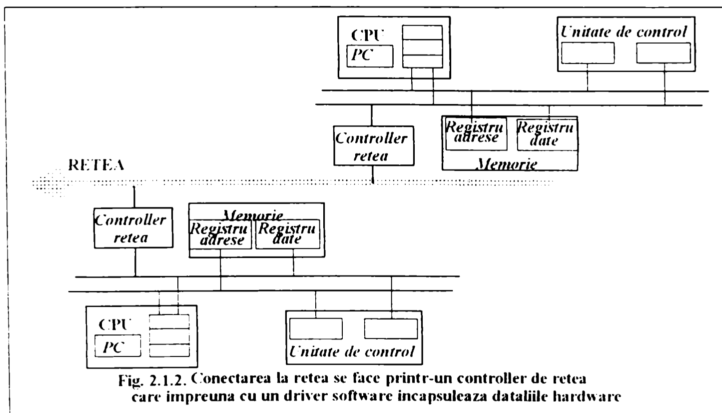


Fig. 2.1.2. Conectarea la retea se face printr-un controller de retea care impreuna cu un driver software incapsuleaza dataliile hardware

Complexitatea sistemului de comunicaţie poate fi redusă prin separarea lui în două mari părţi, fiecare la rindul ei fiind structurată pe nivele: o parte înglobează funcţiile care permit utilizarea mediului fizic de

BUPT

comunicare și cealaltă parte cuprinde funcțiile legate de schimbul de mesaje între procese (fig. 2.1.3), oferind astfel o comunicare virtuala.

Numarul de nivele, numele fiecărui nivel și funcțiile aferente fiecărui nivel difera de la un tip de rețea

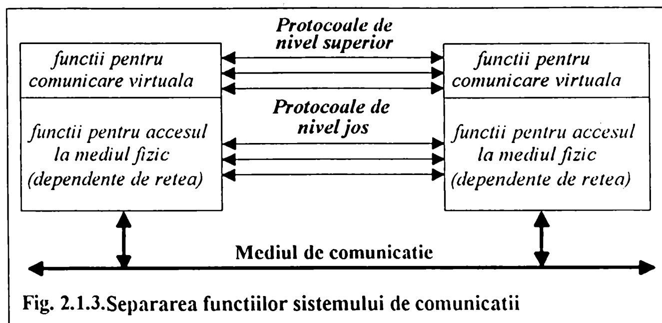


Fig. 2.1.3.Separarea functiilor sistemului de comunicatii

la altul. Interfața dintre două nivele adiacente definește *operațiile primitive* și *serviciile* oferite de nivelul inferior celui superior. Comunicația între nivelele de pe aceleași ranguri este guvernată de un set de reguli, denumite *protocoale;* acestea controlează modul în care se desfasoara comunicația. Un *protocol* poate fi definit ca un set formal de reguli și convenții care guvernează formatul și sincronizarea mesajelor în cadrul unui schimb de mesaje între două sau mai multe procese. Proiectarea unui protocol necesita construirea scenariului de bază în care se definesc mesajele și secvențele de bază din cadrul unui schimb și se specifică procedurile pentru recuperarea erorilor; acestea introduc noi tipuri de mesaje și măresc lista de secvențe de mesaje acceptate într-un schimb.

Protocoalele trebuie să respecte *cerinte cantitative* și *calitative* relative la interconectarea proceselor. Cerintele calitative se referă la mecanisme de sincronizare, detecția și recuperarea erorilor, evitarea interblocarilor, evitarea depășirii capacității buffere-lor, asigurarea secvențierii mesajelor, detectarea mesajelor duplicate, mecanisme pentru asigurarea securității datelor, mecanisme pentru prioritate, mecanisme de contabilizare, garantarea transmiterii mesajelor, transformări de cod/format și compatibilitate cu caracteristicile sistemului de operare și ale mașinii fizice. Cerințele cantitative se referă la întîrzieri de transmisie, productivitatea (cantitatea de date transferate în unitatea de timp), probabilitatea unui eșec în tratarea erorilor. Totalitatea nivelelor și protocoalelor asociate lor definesc *arhitectura rețelei.*

2.1.1. Modelul arhitectural ISO/OSI

Datorita diversificării echipamentelor de calcul, dar și a mediilor de comunicație (rețele locale, rețele de comunicare prin radio sau satelit, rețele digitale cu servicii integrate) și cerinței de interconectare a rețelelor și echipamentelor de proveniență diferită, în scopul alegerii unei variante optime pentru o aplicație, a devenit necesară elaborarea unor standarde internaționale care să guverneze schimbul de informație între calculatoare, rețele, terminale și chiar procese , creindu-se astfel posibilitatea interconectarii unor sisteme *deschise*. Ca atare, Organizatia Internationala de Standarde (ISO) a propus un set de reguli în cadrul unui model de referință pentru interconectarea sistemelor deschise (ISO/OSI RM). Un sistem *deschis* este în acceptiunea ISO OSI un sistem cu o arhitectură standardizată public, care poate să fie conectat și deci să comunice cu orice alt sistem deschis care utilizează același model de arhitectura.

Scopurile propunerii modelului ISO OSI au fost următoarele [Sta89]:(1)-posibilitatea interconectarii sistemelor realizate de diversi producatori; (2)- realizarea unui cadru conceptual coordonarii și definirii de standarde pentru interconectarea sistemelor; (3)-identificarea domeniilor în care sunt necesare dezvoltari și imbunatatiri ale standardelor; (4)-furnizarea unei referințe comune pentru menținerea consecvenței tuturor standardelor.

Setul de protocoale propus de ISO/OSI este stratificat ierarhic pe sapte nivele, fiecare nivel furnizîndu-i servicii nivelului superior, înbunatatind într-un fel calitatea serviciilor oferite de nivelele inferioare (2.1.1.1.a, b). Separarea pe nivele a protocoalelor și serviciilor s-a realizat avîndu-se în vedere obiective precum: (1) un număr cît mai redus de nivele cu puține interacțiuni între ele, astfel încît modificarea funcțiilor unui nivel să nu afecteze celelalte module; (2) comasarea funcțiilor similare în același nivel și în concordanță cu necesitățile de standardizare sau standardele deja în folosință.

Terminologia OSI este ilustrată în fig.2.1.1.2. Într-un sistem există una sau mai multe entități în fiecare nivel. O *entitate* de la nivelul N (entitate (N)) implementează funcțiile nivelului (N) și protocolul de comunicație cu entitățile de la nivelul N din alte sisteme. Exemple de entități pot fi *procesele* (într-un sistem multitasking) și *procedurile*. Pot fi mai multe entități într-un nivel dacă acest lucru este eficient pentru un anumit sistem, eventual diferite, fiecare corespunzînd unui protocol standard de la acel nivel. Fiecare entitate

BUPT

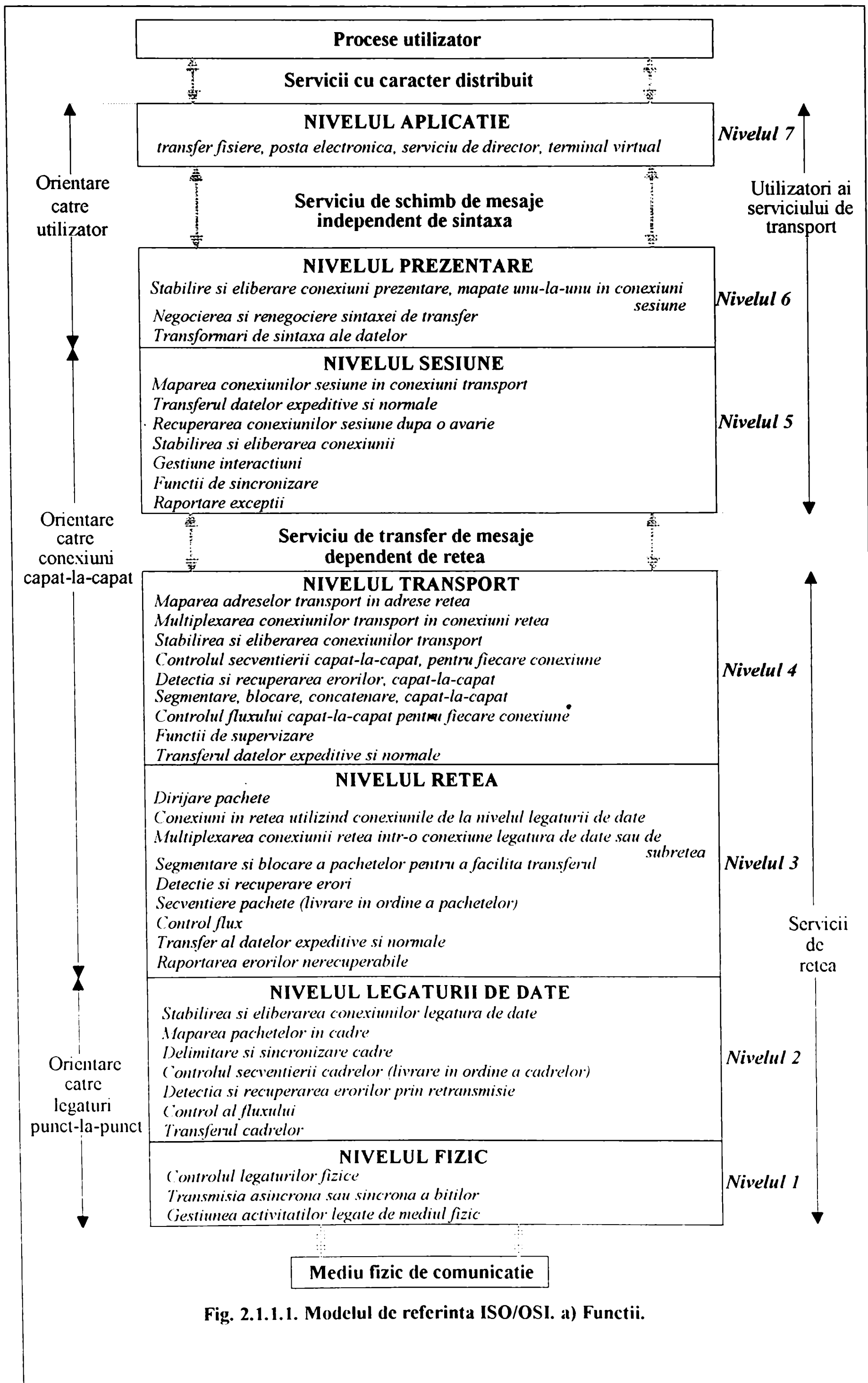


Fig. 2.1.1.1. Modelul de referinta ISO/OSI. a) Functii.

Proces utilizator (aplicatie) — date — Proces utilizator (aplicatie)

Nivel 7 NIVELUL APLICATIE — *Se realizeaza un schimb de informatii intre procese utilizator* — AH | date — NIVELUL APLICATIE Nivel 7

Nivel 6 NIVELUL PREZENTARE — *Datele utilizator pot fi convertite, compactate, cifrate si reorganizate in mesaje de sesiune* — PH | unitate de date — NIVELUL PREZENTARE Nivel 6

Nivel 5 NIVELUL SESIUNE — *Se stabilesc si inchid sesiuni* / *Se recupereaza erorile daca o sesiune este intrerupta* — SH | unitate de date — NIVELUL SESIUNE Nivel 5

Nivel 4 NIVELUL TRANSPORT — *Mesajele sesiune se transmit via subsistemul transport* / *Se pastreaza integritatea mesajului sesiune, garantindu-se ca mesajele nu vor fi pierdute sau dublu receptionate* — TH | unitate de date — NIVELUL TRANSPORT Nivel 4

Nivel 3 NIVELUL RETEA — *Un mesaj poate fi transmis in mai multe pachete. Pachetele pot fi transmise intr-un circuit virtual. Fiecare pachet contine o adresa de destinatie si un numar de secventa.* / *Se transmit pachete de date si pachete de control pentru controlul transmisiei.* / *Cind apar dificultati, pachetele pot urma rute diferite.* — NH | unitate de date — NIVELUL RETEA Nivel 3

Nivel 2 NIVELUL LEGATURII DE DATE — *Se transmit cadre si se primesc confirmari in cadrul unui circuit fizic. In mod obisnuit fiecare cadru contine un pachet.* / *Se detecteaza erorile de transmisie si se retransmit cadrele.* — L Header | unitate de date | L Trailer — NIVELUL LEGATURII DE DATE Nivel 2

Nivel 1 NIVELUL FIZIC — *Se stabileste un circuit fizic in cadrul caruia se transmit biti.* — biti — NIVELUL FIZIC Nivel 1

Mediu fizic de comunicatie

Fig. 2.1.1.1. b) Protocoale.

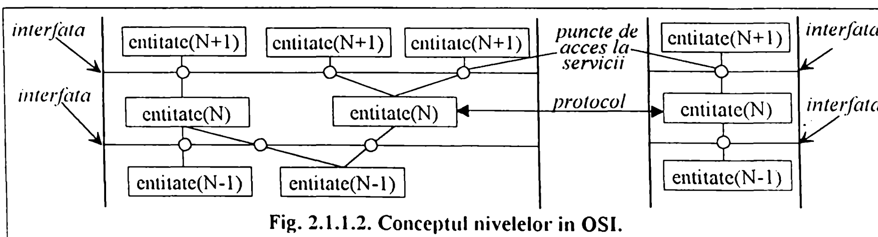
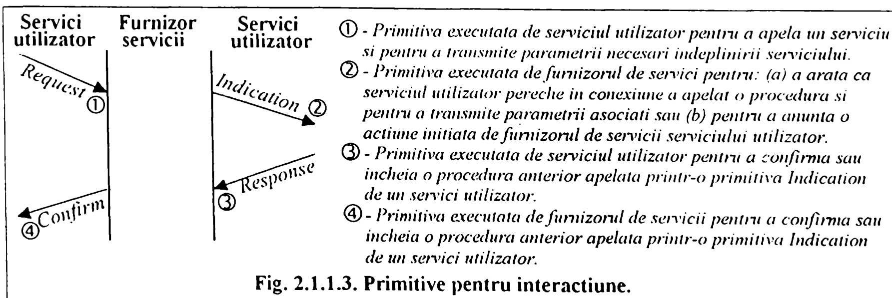


Fig. 2.1.1.2. Conceptul nivelelor in OSI.



① - *Primitiva executata de serviciul utilizator pentru a apela un serviciu si pentru a transmite parametrii necesari indeplinirii serviciului.*
② - *Primitiva executata de furnizorul de servici pentru: (a) a arata ca serviciul utilizator pereche in conexiune a apelat o procedura si pentru a transmite parametrii asociati sau (b) pentru a anunta o actiune initiata de furnizorul de servicii serviciului utilizator.*
③ - *Primitiva executata de serviciul utilizator pentru a confirma sau incheia o procedura anterior apelata printr-o primitiva Indication de un servici utilizator.*
④ - *Primitiva executata de furnizorul de servicii pentru a confirma sau incheia o procedura anterior apelata printr-o primitiva Indication de un servici utilizator.*

Fig. 2.1.1.3. Primitive pentru interactiune.

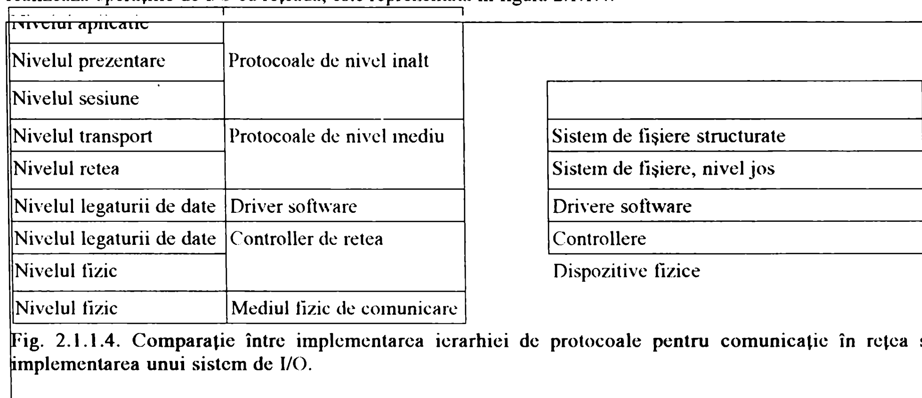
BUPT

comunică cu nivelul superior printr-o interfaţă; aceasta este realizată printr-unul sau mai multe puncte de intrare la servicii (Service Access Points=SAP). Entităţile de la un nivel N furnizează servicii entităţilor de la nivelul (N+1) printr-un set de primitive. O *primitiva* specifică funcţia de executat şi se utilizează pentru a transmite informaţii de control şi date. Forma de realizare a primitivei este dependentă de implementare (un exemplu este un *apel de procedura*). Exista 4 tipuri de primitive utilizate pentru a defini interacţiunea între nivele adiacente ale arhitecturii (fig.2.1.1.3).

Datele se transmit între entităţi pereche sub forma *unităţilor de date ale protocolului (PDU).* O entitate (N) poate transmite o PDU la nivelul N-1, care o percepe ca pe o *unitate de date de serviciu (SDU)* ce trebuie sau nu mapată în mai multe PDU de la nivelul N-1. Dacă maparea este necesară, atunci entitatea pereche (N-1) trebuie să asambleze PDU-urile primite, înainte de a le livra entităţii (N). În fig.2.1.1.1 b se ilustrează principiile OSI relativ la protocoale şi unităţile de date ale protocoalelor.

Arhitectura generală propusă pe sapte nivele de protocoale este un *model de referinţă* care descrie un ansamblu abstract de interdependenţe, un cadru general în care pot fi definite şi dezvoltate standarde, fără a impune restricţii în ceea ce priveşte implementarea, topologia şi tehnologia de interconectare.

O implementare a ierarhiei de protocoale ce pot fi considerate, de fapt, o maşină abstractă ce realizează operaţiile de I/O cu reţeaua, este reprezentată în figura 2.1.1.4.



Nivelul aplicatie		
Nivelul prezentare	Protocoale de nivel inalt	
Nivelul sesiune		
Nivelul transport	Protocoale de nivel mediu	Sistem de fişiere structurate
Nivelul retea		Sistem de fişiere, nivel jos
Nivelul legaturii de date	Driver software	Drivere software
Nivelul legaturii de date	Controller de retea	Controllere
Nivelul fizic		Dispozitive fizice
Nivelul fizic	Mediul fizic de comunicare	

Fig. 2.1.1.4. Comparaţie între implementarea ierarhiei de protocoale pentru comunicaţie în reţea şi implementarea unui sistem de I/O.

Nivelul legaturii de date este parţial implementat în controller-ul de reţea şi parţial într-un driver software. Nivelul fizic este implementat în controller-ul de reţea şi în mediul fizic de comunicare. Protocoalele de nivel mediu (nivelul de reţea şi nivelul de transport) furnizează aplicaţiilor independenţa de hardware, iar cele de nivel înalt extind serviciile puse la dispoziţia programelor.

Facînd o comparaţie între organizarea ierarhică traditională a sistemului de I/O al calculatoarelor (drivere de disc, sisteme de fişiere, etc.) reprezentată în figura 2.1.1.4 şi organizarea ierarhică a sistemului de comunicaţie se poate considera că nivelul nivelul transport, uneori chiar nivelul reţea, corespund nivelului de fişiere, deşi există şi implementari în care, parţial sau total, în special nivelul reţea, este implementat în driver. Se poate spune că, analog sistemului de fişiere care furnizează o abstractizare a operaţiilor de I/O, protocoalele la nivel mediu furnizează o abstractizare a operaţiilor de I/O cu reţeaua de comunicaţii.

2.1.2. Nivele dependente de reţea. Prezentare generală

Functiile nivelelor pot fi descrise pe scurt astfel:

A. ***Nivelul fizic*** defineşte modalitatea în care succesiunea de biţi este transmisa pe mediul fizic de comunicaţie, respectind ordinea biţilor şi fără a garanta o transmisie sigura. Standardele de la acest nivel au în vedere aspecte electrice, functionale, procedurale şi mecanice ale transmisiei pe mediul fizic. Aspectele principale sunt legate de modul de codificare al biţilor, alegerea mediului fizic pentru transmisie (cablu torsadat, cablu coaxial, fibre optice, frecvenţe radio), modul de stabilire şi desfiinţare a conexiunilor fizice, modul de transmisie (duplex, semiduplex, simplex), nivele electrice utilizate pentru transmisie, definirea pinilor şi asignarea lor la conectori, numărul de biţi pe secundă care pot fi transmişi. Standardele definite la acest nivel trebuie să asigure conectarea echipamentelor atît în reţele locale cît şi în reţele municipale şi teritoriale.

B. ***Nivelul legaturii de date*** are ca funcţie principală transformarea canalului de comunicaţie furnizat de nivelul fizic, într-un canal de comunicare sigur, între două noduri adiacente. În acest scop, datele care urmează să fie transmise nivelului fizic, primite de la nivelul superior nivelului legaturii de date, sunt divizate în *cadre*, a caror structură generală este prezentată în figura 2.1.2.1. În general un cadru este compus din: (1) indicator, o succesiune specială de biţi care delimitează începutul şi sfîrşitul unui cadru (si care nu poate apare în interiorul datelor utile); (2) informaţii de control, care pot include tipul cadrului (se disting cadrele de

BUPT

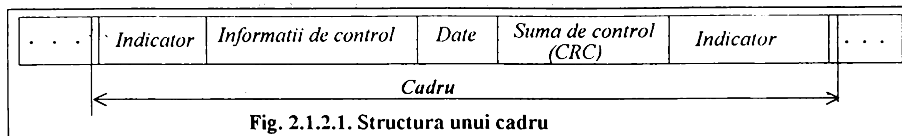


Fig. 2.1.2.1. Structura unui cadru

control de cadrele purtătoare de date utile), numărul de secvență (cadrele se numerotează pentru ca receptorul să poată identifica eventualele cadre duplicate transmise ca urmare a unei erori de time-out), lungimea, adresa sursei și adresa destinației; (3) datele utile; (4) un cîmp CRC care conține suma de control ciclic. Cadrele sunt transmise individual. La recepția lor, se verifică suma de control ciclic si, în cazul unei erori, se poate cere retransmisia cadrului.

O altă funcție a nivelului legaturii de date este *controlul fluxului datelor*, adică a ratei de transmisie a pachetelor dintre două noduri. Controlul fluxului este necesar deoarece: (1) recepția datelor se face în mod obisnuit într-un driver care lucrează cu ajutorul întreruperilor; dacă întreruperile sunt dezactivate în momentul primirii datelor, acestea pot fi pierdute; (2) driverul primeste datele adresate oricărui proces local; ca atare pastrează datele în bufferele sale locale pînă cînd procesul destinatar cere sistemului de operare local să-i livreze un mesaj. Cînd nu mai există spațiu în buffere, driverul nu mai poate recepționa date fără a suprascrie alte date, nelivrate unui proces destinatăr. Pentru contolul fluxului se utilizează frecvent două soluții [Nut92]:

(1) *mecanismul confirmării cadrelor (protocolul stop-and-wait)*, în cadrul căruia transmiterea unui cadru se face numai după primirea unui mesaj de confirmare a cadrului precedent; în termenii comunicației interprocese în sistemele de operare, este analog unui apel send() sincron (fig. 2.1.2.2.). Sincronizarea între receptor și emițător se realizează printr-un cadru special de tip confirmare (ACK), care nu conține date utile. Pentru a preveni așteptarea la nesfarsit a unei confirmări "pierdute" sau a confirmării unui cadru "pierdut" (în ambele cazuri se presupune că nu a ajuns la destinație cadrul) se folosește un mecanism de time-out și se numerotează cadrele.

```
Emitatorul_transmite_cadrul();
Emitatorul_seteaza_un_timer_la_valoarea_TIMEOUT_pentru_a_urmari_transmisia();
Emitatorul_asteapta_o_intrerupere_de_receptie_cadru_sau_de_time-out();
if (Emitatorul_a_primit_ACK) {
        Emitatorul reseteaza timerul();
        continue;          }
if (Timerul_a_generat_TIMEOUT)
        Retransmite_cadrul_care_a_produs_time-out();
```

(a) Secventa de cod executata pentru transmisie

```
Receptorul_accepta un cadru neduplicat();
Receptorul_transmite_un_cadru ACK();
```

(b) Secventa de cod executata pentru receptie

Fig. 2.1.2.2.Controlul fluxului prin protocolul stop and wait

```
(1) if( Es-Ei<dimensiunea_maxima_a_ferestrei ) {
        Emitatorul transmite_cadrul_Es ();
        Emitatorul_seteaza_un-timer la-valoarea TIMEOUT pentru_a_ urmari transmisia acestui cadru ();
        Incrementeaza_Es;
    }
(2) if( Emitatorul_a_primit_ACK ) {
        Emitatorul reseteaza timerele pentru cadrele de dinaintea_celui pentru care a primit ACK ();
        Incrementeaza_Ei;
    }
(3) if( un-timer a generat TIMEOUT ) {
        Reseteaza Es, Ei;
        Retransmite toate_cadrele de dupa_cel care-a_generat_time-out ()
    }
```

(a) Secventa de cod executata la transmisie:
1) la primirea unui cadru pentru transmisie;
2) la generarea unei intreruperi de receptie de la driver;
3) la generarea unei intreruperi de time-out de un timer.

```
Receptorul_accepta-un_cadru neduplicat ();
if( Ri == numarul_de secventa al cadrului ) {
      Incrementeaza_Ri_mod dimensiunea-maxima a ferestrei + 1 ();
      Receptorul transmite_confirmarea-ACK_pentru acest cadru ();
}else Rejecteaza cadrul ();
```

b) Secventa de cod executata la receptie

Fig. 2.1.2.3. Controlul fluxului prin protocolul cu fereastra glisanta.

(2) *protocolul cu fereastra glisantă (sliding window protocol)* este o generalizare a celui de mai sus, în sensul că admite ca emițătorul să aibă N cadre în tranzit *înainte* de primirea unei confirmări. În acest caz, dacă se numerotează cadrele, emițătorul va lucra cu doi pointeri, Ei și Es, care pointează la cadrul cu numărul

BUPT

cel mai mic care a fost transmis, dar pentru care nu s-a primit confirmare, respectiv la primul cadru netransmis; astfel, toate cadrele cu numerele între Ei şi Es, care alcătuiesc fereastra curentă a emiţătorului, sunt în tranzit. Protocolul permite "pierderea" confirmărilor unui cadru, deoarece confirmarea cadrului *i* provoaca automat confirmarea cadrelor *i-1, i-2,...Ei.* La emisie, se va folosi pentru fiecare cadru neconfirmat cîte un timer. Emitatorul va putea avansa cu maximum N cadre fără a primi o confirmare; cînd se atinge limita ferestrei, se întrerupe transmisia cadrelor pînă la primirea unei confirmări sau a generării unui semnal de time-out.

Există şi o altă variantă a protocolului, în care, în cazul apariţiei unei erori time-out nu este necesar să se retransmita toate mesajele din bufferul emiţătorului, ci numai mesajul nerecepţionat corect (sau neconfirmat); aceasta presupune că şi receptorul să lucreze cu o fereastra de recepţie. Trebuie însa să fie mai mică decît numărul maxim de secvenţă al cadrelor (N); în general are valoarea (N+1)/2.

C. ***Nivelul reţea*** are drept scop final crearea posibilităţii de distribuţie a informaţiei în reţele logice, în care adresele să fie generalizate, fără a depinde de spaţiile de adrese specifice diferitelor tipuri de protocoale de la nivelul legaturii de date. Pentru a realiza acest scop, la nivelul reţea trebuie să se realizeze interconectarea reţelelor cu arhitecturi diferite şi dirijarea pachetelor de date între nodul sursa şi cel destinaţie, indiferent dacă acestea sunt sau nu adiacente (se vor folosi noduri intermediare).

Adresele din pachete sunt interpretate de protocoalele de la nivelul reţea; acestea utilizează un spaţiu de adrese diferit de cel utilizat de cadrele nivelului legaturii de date; un cadru trebuie să specifice ca adresă destinaţie numai adresa unui nod adiacent, pe cind, la primirea unui pachet, nivelul reţea poate să transmită mai departe pachetul, încapsulîndu-l într-un cadru (fig 2.1.2.4), permiţînd crearea unui efect logic de conectare a tuturor nodurilor mai multor reţele interconectate. Rezulta că nivelul reţea trebuie să furnizeze un mecanism uniform de adresare care să aiba în vedere diversitatea reţelelor de comunicaţie şi faptul că acestea pot fi distribuite geografic pe ţări, continente şi aflate în proprietatea sau gestiunea unor autorităţi diferite.

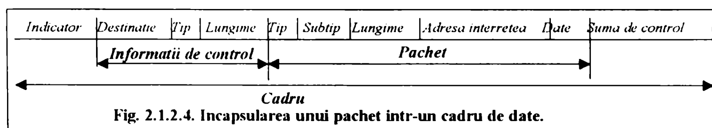


Fig. 2.1.2.4. Incapsularea unui pachet intr-un cadru de date.

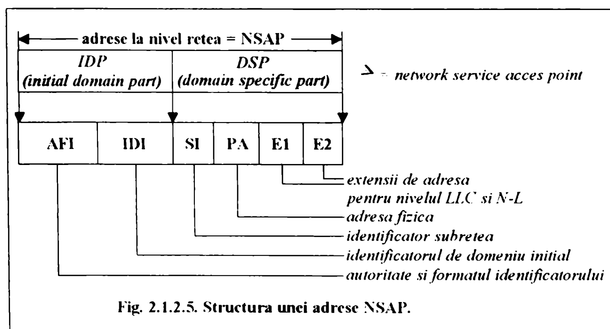


Fig. 2.1.2.5. Structura unei adrese NSAP.

Pentru adresare la nivelul reţea, ISO a propus o structură ierarhizată a adreselor, reprezentată în figura 2.1.2.5. Fiecare adresa este compusă din două părţi: *partea de domeniu initial ("Initial domain part" IDP)* şi *partea specifică domeniului ("domain specific part"- DSP).* Pentru a reflecta numeroasele alternative de interconectare a reţelelor de diferite tipuri, fiecare cu propria sa schema de adresare, cîmpul IDP este la rîndul lui impărţit în două părţi: *autoritatea* şi *identificatorul formatului* (AFI) şi *identificatorul formatului initial* (IDI). Campul AFI specifică atît modul de interconectare al reţelelor, dar şi formatul cîmpului IDI asociat; de exemplu, dacă interconectarea se face prin reţele internationale cu comutare de pachete, IDI identifică adresa porţii locale cu care subreţeaua este conectată şi va cuprinde o secvenţă de maximum 14 cifre alocate în conformitate cu recomandarea CCITT X.121, iar cîmpul AFI va conţine valoarea zecimală 36 dacă adresa IDI este reprezentată în forma BCD sau valoarea zecimală 37, dacă IDI este reprezentată în forma binară. În tabelul din figura 2.1.2.6 se arată şi alte valori pentru AFI. Pentru toate aceste valori, adresa IDI este compusă dintr-un număr de alte subadrese ierarhice care reprezintă codul tarii şi codul naţional al reţelei. Campul DSP cuprinde, analog, un număr de subadrese ierarhice: identificatorul subreţelei (SI), care identifică subreţeaua particulară (LAN), la care este conectată statia destinaţie, adresa fizică (PA) care precizează adresa fizică a subreţelei identificată de SI.

BUPT

Formatul IDI	Valoare AFI		Lungime IDI (cifre)	Lungime DSP (cifre zecimale)
	Zecimal	Binar		
Retele internationale cu comutare de pachete X.121	36	37	14	24
Retele telex internationale F.69	40	41	8	30
Retele internationale telefonice comutate E.164	42	43	12	26
Retele internationale digitale cu servicii integrate	44 48	45 49	15 Nula	23 38

Fig. 2.1.2.6. Valori pentru cîmpurile unei adrese NSAP

Dirijarea pachetelor este una dintre funcţiile esentiale ale nivelului reţea care, indiferent de algoritmul folosit pentru dirijare, se poate rezuma la următoarele operaţii: (1) nodul emiţător trebuie să utilizeze o tabela de dirijare (locală), cu ajutorul căreia să poată selecta un prim nod intermediar în calea către destinaţie; (2) nodul emiţător trimite pachetul încapsulat într-un cadru nodului intermediar; (3)nodul intermediar determină dacă pachetul trebuie dirijat în continuare către destinaţie, utilizînd propria să tabela de rutare sau a ajuns la destinaţie. Algoritmii de dirijare a pachetelor, care trebuie să satisfaca cerinţe ([Tan88] [And94]) de simplitate, robustete, stabilitate, optimalitate, nepărtinire (algoritmul nu trebuie să favorizeze o categorie de utilizatori în defavoarea celorlalti, cu excepţia cazului cînd se atribuie priorităţi), pot fi clasificaţi, după reacţia la modificările traficului sau topologiile reţelei, în algoritmi *adaptivi şi neadaptivi*. Din clasa algoritmilor neadaptivi, care realizează o dirijare statică se enumeră *dirijarea fixa* (in variantele dirijarea cea mai scurta şi dirijarea multicai), *dirijarea în flux, dirijarea incidentală şi dirijarea ierarhica*, iar din clasa algoritmilor adaptivi: *dirijarea centralizată, dirijarea izolată* (algoritmii *"hotpotato"* şi *"backward learning"*), *dirijarea delta şi dirijarea distribuita*. Pentru reţelele de largă întindere geografică este importantă dirijarea ierarhică.

Indiferent de algoritmul de dirijare utilizat, trebuie evitată degradarea performanţelor reţelei, ca urmare a supraîncărcării ei. În acest scop, o altă funcţie a nivelului reţea este *evitarea congestionarii reţelei*; ea poate apare datorita dispariţiei unei legături în reţea, a tranziţiei la noi tabele de dirijare sau a sufocării unor noduri utilizate pentru a calcula căi de dirijare. Exista două categorii de soluţii pentru a evita congestionarea reţelei: (1) evitarea supraîncărcării fiecărui nod şi (2) menţinerea traficului în jurul unei nivel fixat pe liniile de legătură dintre două procese utilizator (pe calea unui circuit virtual), pe toate liniile de legatură din jurul unui nod sau pe liniile de legatură dintre noduri şi comutatoarele de pachete. Din algoritmii de categoria (1), de control a supraîncărcării fiecărui nod, se enumeră ([For88][Tan88]): prealocarea zonelor tampon în cadrul unui circuit virtual, ignorarea pachetelor pentru care nu există spaţiu pentru memorare şi utilizarea unor pachete de permisiune (in număr limitat în reţea) pentru a putea transmite pachete de date. Dintre algoritmii de categoria (2), de control al fluxului de mesaje, se specifică doi algoritmi: transmiterea de către comutatoarele de pachete a unor pachete de şoc nodurilor sursa, cu scopul de a le impune micşorarea ratei de transfer a pachetelor pe liniile din jurul comutatoarelor de pachete (care îşi monitorizează activitatea liniilor de ieşire) şi impunerea transmisiei unui pachet numai după primirea confirmării recepţiei celui transmis anterior.

O altă situaţie care trebuie luată în considerare la nivelul reţea este *blocarea definitiva a nodurilor*; aceasta poate apare în diverse situaţii, cele mai posibile fiind: (1) cînd două comutatoare de pachete trebuie să-si transmita reciproc cîte un pachet, dar ambele au toate bufferele ocupate sau (2) cînd un comutator de pachete, care foloseşte la nivelul legatură de date un protocol de fereastra glisantă, varianta selectivă, (in care se memorează în buffere mesajele recepţionate corect, în limita ferestrei, chiar dacă s-a pierdut un mesaj anterior) iar pentru alocarea bufferelor sale locale la circuitele virtuale foloseşte o strategie FIFO globală, are toate bufferele locale ocupate cu pachete provenind, pentru un anumit circuit virtual, de la acelaşi mesaj, cu numerele de secvenţă mai mari decit ale altor pachete aşteptate pentru care nu mai există loc în buffere. Dintre soluţiile utilizate pentru evitarea blocarii definitive se enumeră: (1) utilizarea în fiecare nod a M+1 buffere, M fiind lungimea maxima a căilor reţelei; (2) asocierea unei informaţii de vechime fiecărui pachet;
(3) transmisia pachetelor obţinute prin fragmentarea unui mesaj numai după alocarea unui număr suficient de buffere în care să se poată memora pachetele mesajului multipachet.

Unul dintre cele mai discutate aspecte ale proiectarii nivelului reţea a fost *tipul* serviciilor care trebuie furnizate de acest nivel nivelului sau superior, cel de transport: *servicii orientate* sau *neorientate pe conexiune*. Problema poate fi extinsă şi în cazul reţelelor locale: ce tip de serviciu trebuie să ofere nivelului transport nivelul legaturii de date (eventual, prin intermediul nivelului de reţea, dacă acesta există). Pe de altă parte, deoarece arhitectura modelului OSI pentru subreţeaua de comunicaţii prevede ca ultim nivel nivelul reţea, rezultă că serviciul oferit de acesta este serviciul oferit de subreţea în general. Ca atare, ISO a hotărît ca modeul OSI să ofere la nivelul reţea ambele tipuri de servicii: (1) servicii orientate pe conexiune; (2)servicii neorientate pe conexiune.

BUPT

Tipul (1) se poate caracteriza asfel [Tan88][Gos91]: (1) Între 2 entităţi care comunică, la cererea sursei, se stabileşte mai întîi un canal logic sau o conexiune. Această conexiune primeşte un identificator care ramîne valabil pînă la terminarea datelor de transmis şi desfiinţarea explicită a conexiunii, de către oricare dintre cele două entitati. (2) La stabilirea unei conexiuni, cele două entităţi de la nivelul transport şi nivelul reţea care furnizează serviciul stabilesc *parametrii* (adresele sursei şi destinaţiei, şi indicatori pentru utilizarea sau nu a confirmărilor şi a datelor expeditive) şi *calitatea serviciilor* în cursul unui proces de negociere. (3) Comunicaţia este permisa în ambele sensuri, pastrîndu-se ordinea pachetelor, garantîndu-se transmisia şi controlul fluxului datelor astfel încît comunicaţia este posibila şi între calculatoare de performanţe diferite. Pentru tipul (1) de servicii sunt prevăzute primitive pentru stabilirea conexiunii pentru trimiterea şi recepţionarea datelor şi pentru desfinţarea conexiunii.

În contrast, *serviciile neorientate pe conexiune* transmit individual pachete care, nefiind integrate într-o conexiune, trebuie să încorporeze, întotdeauna, adrese destinaţie; nu se garantează sosirea în secvenţă a pachetelor. Pentru acest tip de servicii sunt prevăzute primitive numai pentru transmiterea şi recepţionarea datelor.

Aspect	*Stabilire conexiune*	*Adresa destinatie*	*Secventiere pachete*	*Control erori*	*Control flux*	*Este posibila negocierea de optiuni*	*Se utilizează identificatori de conexiune*
Serviciu orientat pe conexiune	Necesar	Numai la stabilirea conexiunii	Garantata	Realizat la nivelul retea	Realizat la nivelul retea	Da	Da
Serviciu neorientat pe conexiune	Imposibil	La orice pachet	Negarantata	Realizat la nivelul transport	Nu este furnizat de nivelul retea	Nu	Nu

Fig. 2.1.2.7. Principalele diferente între serviciile orientate pe conexiune şi cele neorientate pe conexiune

În figura 2.1.2.8. se prezintă repartizarea celor două tipuri de servicii (orientate şi neorientate conexiune) pe toate nivelele OSI. Se observă că este posibil să se dispună de servicii orientate conexiune la nivelul reţea sau transport, chiar cînd nivelele inferioare nu sunt orientate pe conexiune. De exemplu, un serviciu orientat pe conexiune la nivelul transport poate fi construit într-o reţea locală pe baza unui protocol neorientat pe conexiune tratînd controlul erorilor, controlul fluxului, stabilirea conexiunilor la nivelul reţea sau transport.

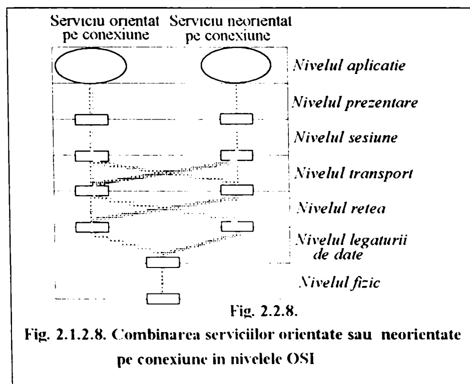


Fig. 2.1.2.8. Combinarea serviciilor orientate sau neorientate pe conexiune in nivelele OSI

Ambele tipuri de servicii au avantaje şi dezavantaje. Astfel, serviciile neorientate pe conexiune sunt mai rapide (nu este necesara stabilirea conexiunilor), au protocoale simple, dar şi dezavantajele că nu asigură contolul fluxului şi livrarea în secvenţă a pachetelor, erorile (pachete pierdute sau duplicate) trebuie tratate la nivelul superior, iar adresa destinaţie încarcă toate pachetele. Pe de altă parte, deşi dezavantajele de mai sus sunt înlăturate la serviciile orientate pe conexiune, acestea sunt puţin eficiente, datorita regiei de sistem sporite.

În cazul reţelelor cu comutarea pachetelor, organizarea subreţelei de comunicaţie poate fi facută folosind *tehnica circuitelor virtuale* sau a *datagramelor*. *Tehnica circuitelor virtuale* se utilizează, în general, în legatură cu serviciul orientat pe conexiune, deoarece presupune dirijarea tuturor pachetelor dintre un nod sursa şi un nod destinaţie pe aceeaşi cale, fixată iniţial printr-un pachet de stabilire a circuitului; de aceea, pe de o parte, în antetul pachetului se precizează numărul circuitului şi nu adresa completă a destinatarului şi, pe de altă parte, fiecare nod de comunicaţie păstrează un tabel cu circuitele virtuale care îl parcurg.

Tehnica datagramelor se utilizează în legătură cu serviciul neorientat pe conexiune, deoarece presupune dirijarea pachetelor individual, independent de calea urmată de pachetele anterioare între o aceeaşi sursă şi o aceeaşi destinaţie; în acest caz, fiecare pachet trebuie să conţină în header adresa completă a destinaţiei, irosindu-se mai mult spaţiu decît numărul circuitului virtual din cealaltă tehnică.

BUPT

Dèşi nu se conformează modelului OSI, este posibil ca unele aplicaţii să fie construite direct deasupra nivelului reţea; în acest caz, ele percep reţeaua ca un mediu de comunicaţie nesigur, şi de aceea trebuie să trateze erorile (de pierdere sau duplicare a pachetelor) şi să controleze fluxul de date şi primirea în secvenţă a pachetelor. Exista categorii de aplicaţii (de exemplu, aplicaţiile legate de voce), pentru care este preferat serviciul neorientat pe conexiune, fiind mai rapid, chiar dacă nu garantează 100% transmisia fără erori a pachetelor; în acest caz, pierderea unui mesaj nu este catastrofă (destinaţia se poate dispensa de datele conţinute într-un pachet pierdut sau poate să le determine printr-un mecanism de interpolare).

O altă funcţie a nivelului reţea, în conceptia ISO, este *interconectarea reţelelor* (LAN-LAN, LAN-WAN, WAN-WAN, LAN-WAN-LAN); aceasta impune următoarele cerinţe: (1) suport pentru realizarea legaturilor fizice între reţele conectate; este necesar, în acest caz, la minimum, nivelul fizic şi nivelul legaturii de date; (2) suport pntru dirijarea pachetelor între procese aflate în noduri aparţinînd unor reţele de tipuri diferite; (3) suport pentru un serviciu de monitorizare a utilizarii reţelelor interconectate şi a dispozitivelor de interconectare (porţi) şi pentru memorarea unor informaţii de stare; (4) satisfacerea cerinţelor de mai sus în condiţiile în care nu sunt permise modificari ale arhitecturii modelului OSI sau ale reţelelor componente. Aceasta înseamnă că interconectarea reţelelor trebuie să rezolve problema diferenţelor multiple dintre reţelele conectate, precum: scheme de adresare diferite, diferenţe între dimensiunea pachetelor (divizarea unui pachet mai mare provenind dintr-o reţea în pachete mai mici pentru a putea fi trimise în altă reţea este numita *segmentare*), mecanisme de acces la reţea diferite, valori diferite pentru timeout, tehnici de dirijare diferite; parametrii pentru raportarea starii şi performanţelor diferiţi, utilizarea circuitelor virtuale într-o reţea şi a datagramelor în alta. Complexitatea soluţiilor de interconectare a reţelelor, în general (nu neaparat OSI) este prezentată în figura 2.1.2.9.

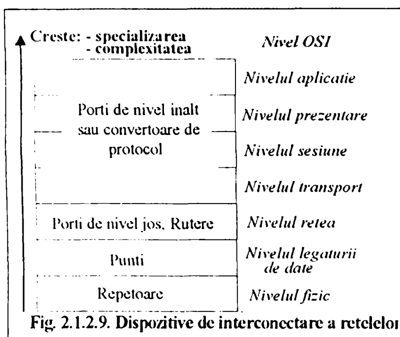


Fig. 2.1.2.9. Dispozitive de interconectare a retelelor

În conceptia OSI ([Sta89][Tan88]), pentru a putea realiza o funcţie de interconectare a reţelelor, acolo unde este nevoie, nivelul reţea este impărţit în 3 subnivele: (a) subnivel de acces la reţeaua de comunicaţii, cu rolul de arealiza funcţiile nivelului reţea în conformitate cu protocolul subreţelei conectate; (b) nivelulul de imbogatire a reţelei, cu rolul de a uniformiza deosebirile dintre subreţelele conectate; (c) nivelul de interconectare a reţelelor, cu rolul de a realiza funcţiile nivelului reţea in subreţeaua de comunicaţie (în special dirijarea pachetelor - cu cele două variante, datagrame sau circuit virtual - preferîndu-se, de obicei un algoritm de dirijare ierarhica).

Proiectarea *software-ului de comunicaţie al punţilor*, care conectează reţelele locale, ca şi al porţilor, trebuie să primeasca o atenţie specială datorita restrictiilor de timp care se pun. Desi rapida, varianta în care software-ul este organizat pe *întreruperi (interrupt driven)* - fiecare recepţie a unui pachet provoaca declansarea unei întreruperi -, fiind nestructurată, este mai puţin preferabila variantei în care software-ul este organizat pe *procese* sau şi mai bine pe *fire*, tinînd seama că sistemele de operare actuale permit lucrul cu fire. În figura 2.1.2.10. se prezintă organizarea software-ului unei porţi GA care conectează trei reţele. În acest caz, software-ul de comunicaţie este organizat în 9 procese. Procesele de intrare I1, I2, I3, avînd prioritatea cea mai mare, pentru a nu se pierde pachete, sunt executate la recepţia unui pachet: ele au rolul de a verifica suma de control a pachetului şi de a-l depune într-o coade de mesaje asociată procesului R, procesul care realizează dirijarea pachetelor. Procesul R este executat atunci cînd nu este activ nici un proces de prioritate mare (I1, I2, I3), sau de prioritate medie (O1, O2, O3) şi cînd coada de mesaje asociată nu este vidă. În acest caz, rolul său este de a prelua primul pachet din coada, de a determina calea pe care trebuie dirijat, de a-l depune în coada corespunzătoare unuia dintre procesele de ieşire (O1, O2, O3). In momentul activarii, un proces de ieşire preia din coada sa un pachet şi iniţiaza transmisia pachetului, după care se autoblochează; este reactivat la declansarea întreruperii de sfîrşit de transmisie, moment în care incearcă transmisia următorului de pachet din coadă, s.a.m.d. Procesul R are şi rolul de a filtra pachetele: astfel, pachetele de control nu sunt transmise mai departe. Procesele S şi M, de prelucrări statistice şi de gestiune sunt executate periodic, cu cea mai mică prioritate.

Software-ul de comunicaţie expus mai sus, bazat pe conceptul de proces este structurat, dar cu o eficienţă relativ scazută. Pentru a-i creşte eficienţa, pot fi aduse următoarele îmbunătăţiri: (1). Cozile de pachete (structuri de tip FIFO), prin care comunică procesele trebuie implementate, dacă sistemul de operare permite, in zone de memorie partajate de toate procesele care trebuie să aiba acces la ele: se ocoleşte în acest fel copierea mesajelor dintr-un spaţiu de adrese al unui proces în spaţiul de adrese al altui proces. (2). În locul proceselor, se pot folosi fire; acestea prezintă avantajul că nu se iroseşte timp procesor pentru comutarea proceselor şi partajează structurile de date. (3). Se evită complicaţiile care pot apare datorită diferenţei dintre dimensiunea pachetelor (si header-elor) în reţele diferite. O soluţie este de a se recepţiona pachetele în

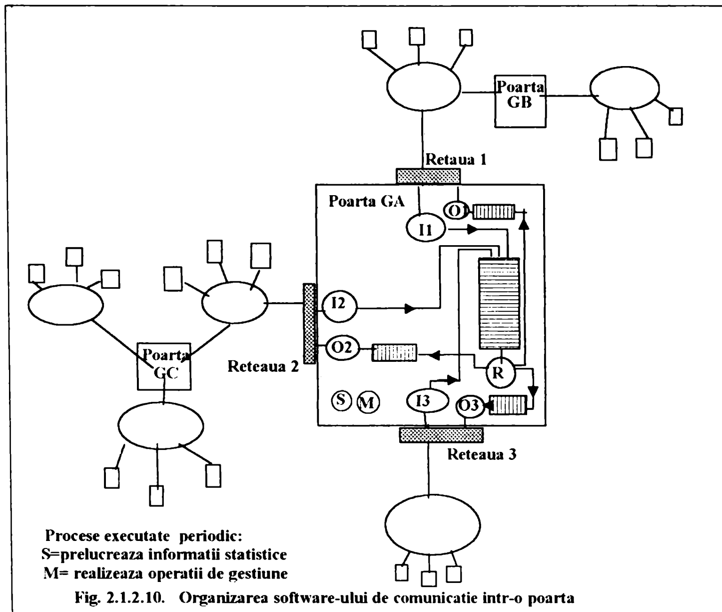


Fig. 2.1.2.10. Organizarea software-ului de comunicatie intr-o poarta

buffere, începînd nu de la adresa de start, ci de la o oarecare distanţă, astfel încît bufferul să poată conţine şi cel mai mare header, în locul celui recepţionat. Figura 2.1.2.11 prezintă un exemplu în care se presupune că cele trei reţele au dimensiunea header-elor de 24 octeti, respectiv 16 şi 8; în acest caz, pachetele din reţeaua 3 vor fi recepţionate în buffer cu un deplasament de 16 octeti faţă de adresa sa de start (fig. 2.1.2.10.a), astfel încît, dacă pachetul va trebui transmis în reţeaua 1, cu un header de 24 octeţi, va exista loc în buffer pentru construcţia unui header mai mare (fig. 2.1.2.10.b). (4) Un alt aspect este *gestiunea timer-elor.* Deşi în funcţie de cerinţele de complexitate a porţii, este necesar, în majoritatea situaţiilor, să se aştepte o confirmare, pînă la eliberarea unui pachet dintr-un buffer. Deoarece în majoritatea cazurilor, pachetele sunt livrate corect la destinaţie, multe dintre timere nu vor declansa timeout. Este important, deci, că setarea timerelor să nu necesite mult timp. Mai multe timere (fizice) pot fi implementate logic, folosind acelaşi timer fizic (fig. 2.1.2.11) şi o listă înlanţuită de cereri de time-out. La fiecare tick al timer-ului se decrementează contorul de timp din nodul curent al listei; cînd acest contor devine 0, se declanşează o întrerupere de time-out şi se elimină nodul din listă, trecîndu-se la tratarea nodului următor. Metoda are avantajul unor operaţii puţine la un tick, dar dezavantajul parcurgerii listei la setarea şi resetarea unui timer. (5) În unele situaţii, de exemplu, în transimisia între reţele 802.3 şi 802.5 este necesară inversarea ordinii biţilor într-un octet. Cea mai simplă soluţie este cea hardware, prin conectarea bitului de intrare 0 la bitul de ieşire 7, bitul 1 la bitul 6, s.a.m.d., dar dacă acest lucru nu este posibil se poate folosi o tabelă cu 256 intrari, indexată după valorile unui octet (a i-a intrare reprezintă octetul de valoare i avînd inversată ordinea biţilor).

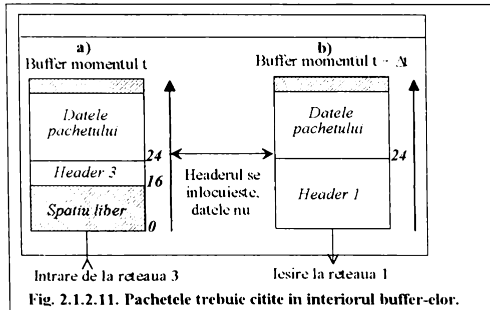


Fig. 2.1.2.11. Pachetele trebuie citite in interiorul buffer-elor.

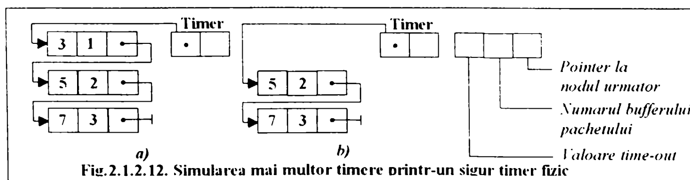


Fig.2.1.2.12. Simularea mai multor timere printr-un sigur timer fizic

BUPT

2.1.3. Nivelul transport, principalul furnizor de servicii pentru sistemele de operare

Se apreciaza în [Tan88][Sta89][Mul89][Nut92] că nivelul transport, nivelul superior al protocoalelor de nivel mediu este centrul de greutate în ierarhia de protocoale. Este considerat interfaţa dintre nivelele dependente de aplicaţie şi nivelele dependente de reţea, cerinţele de deasupra acestui nivel focalizîndu-se pe cerinţe specifice utilizatorilor şi aplicaţiilor. Scopul său este de a furniza nivelului superior un serviciu de transfer sigur al datelor (fără date eronate, duplicate sau pierdute şi respectînd ordinea în care au fost emise), indiferent de tipul reţelei; el maschează astfel detaliile de implementare ale operaţiilor de comunicaţie în reţea punînd la dispoziţia nivelului sesiune o interfaţă relativ simplă care înglobează toate facilităţile de transfer de mesaje. Fara nivelul transport nu ar avea sens toată ierarhia de nivele şi protocoale deoarece scopul final al unei arhitecturi stratificate este de a furniza utilizatorului (şi aplicaţiilor) servicii de transfer de mesaje independente de tipul reţelei utilizate, sub forma unui set standard de primitive.

Un aspect important legat de acest lucru este faptul că, de foarte multe ori, utilizatorii preferă să-şi scrie aplicaţiile folosind direct primitive ale nivelului transport, nefiindu-le necesare serviciile nivelelor de deasupra nivelului transport. Un exemplu sugestiv este cazul sistemului de operare Unix care nu implementează, pentru lucrul în reţele Unix, nivelele de deasupra nivelului de transport. Aplicaţiile Unix folosesc operaţiile de I/O cu reţeaua, analog operaţiilor de lucru cu fişiere, scriind şi citind dintr-o reţea ca într-o conductă *pipe* dar folosind protocolul nivelului de transport; acest mod de lucru a fost ulterior adoptat şi de Microsoft în Windows NT şi Windows 95; mai mult chiar, se poate lucra cu structuri de tip *remote pipe* folosind protocolul de transport.

Un alt aspect important este faptul că nivelul transport este implementat de obicei în interiorul sistemelor de operare, spre deosebire de nivelele inferioare lui care sunt implementate sub forma de drivere software.

Un ultim aspect care trebuie reliefat este faptul că studierea nivelului de transport OSI este utilă şi pentru situatţa în care trebuie proiectate aplicaţii care folosesc direct nivelul reţea (este cazul reţelelor locale) deoarece pot fi găsite soluţii pentru problemele care trebuie rezolvate în acest caz la nivelul aplicaţiei (controlul fluxului datelor în special).

Functiile nivelului de transport sunt legate de adresare, realizarea transferului datelor, segmentarea mesajelor primite la nivelul sesiune, respectarea ordinii de emisie a mesajelor şi la recepţie, controlul erorilor la recepţionarea mesajelor, controlul fluxului mesajelor capăt la capăt, stabilirea şi desfiinţarea cenexiunilor şi multiplexarea conexiunilor. Ca şi la nivelul reţea, sunt posibile două tipuri de servicii: *orientate şi neorientate pe conexiune*, dar sunt caracteristice serviciile orientate pe conexiune. În ambele tipuri de servicii, utilizatorul nivelului transport trebuie să identifice utilizatorul nivelului transport pereche printr-o adresă de nivel transport (*adresa TSAP= transport service acces point*)

Un utilizator al nivelului transport înseamnă un proces în cadrul unui sistem de operare (se consideră că nodurile reţelei sunt gestionate de sisteme de operare multitask), şi deci comunicaţia asigurată de nivelul transport este o comunicaţie între două procese: un proces primar care apelează un proces secundar dintr-un alt nod. De aceea, la nivelul transport se specifică o adresă ierarhică formată din adresa de reţea, adresa de nod şi un port de comunicaţie pentru proces (reţea, nod, port). Modul în care procesul se conectează la port este specific sistemului de operare local (un exemplu este sistemul Unix, în care portul de comunicaţii corespunzător unui proces este stabilit printr-un *socket)*. Adresa de reţea şi adresa de nod vor fi transmise mai departe nivelului reţea, sub forma unei *adrese NSAP* care identifică unic nodul pereche iar *portul* va fi transmis ca informaţie în cadrul pachetului de date (la stabilirea unei conexiuni pentru serviciile orientate pe conexiune şi respectiv în fiecare pachet de date pentru serviciile neorientate pe conexiune). Pentru determinarea portului de comunicaţie corespunzător procesului secundar (care fiind apelat pentru îndeplinirea unui serviciu se numeste *server*), un proces primar (numit *client*, fiind apelant al unui serviciu) poate folosi următoarele metode [Sta89][Tan88]:

1. Procesul primar cunoaste adresa completă corespunzatoare procesului secundar; aceasta este fie (a) o caracteristică de sistem, fie (b) este o adresă facută publică în reţea (*well-known-addresse*) ca aparţinînd unui proces server ale cărui servicii sunt uzuale. Pentru cazul (a) poate fi dat următorul exemplu: în fiecare nod al unei reţele există cîte un proces care colectează informaţii statistice; periodic, un proces master, situat pe un anumit nod, centralizează aceste informaţii conectîndu-se la toate procesele de colectare din celelalte noduri. Porturile de comunicare ale acestora nu trebuie să fie publice, dar pot fi fixate şi cunoscute de procesul master. În cazul (b) pot fi incluse toate serverele care oferă servicii uzuale într-o reţea, transfer de fişiere (ftp, deschiderea de sesiune de lucru de la distanţă-Telnet).

2. În ambele cazuri, 1.a şi 1.b procesul secundar trebuie să existe, să fie activ şi să asculte la adresa sa TSAP fixată, în vederea primirii unei cereri. În cazul (b) cînd numărul de servere cu adrese publice în reţea devine prea mare este ineficient să se păstreze active aceste servere. În acest caz poate fi folosită o altă

BUPT

metoda: procesele server sunt create și activate numai la primirea unei cereri de către un proces server master. Acest server master va fi activ în orice nod în care pot fi activate servere pentru servicii uzuale; el va aștepta cereri de conectare la aceste servere la o adresă TSAP1 facută publică. Cînd va primi o cerere pentru un server, va crea procesul server, transmițîndu-i o adresă TSAP2 la care aceasta să se conecteze pentru a prelua cererile și va transmite această adresă clientului, după care se va deconecta de la client, așteptînd alte cereri de conectare. După stabilirea în acest mod a conexiunii dintre client și server, serverul execută cererile primite de la client iar la desființarea conexiunii de către client se autodistruge. Succesiunea de operații de mai sus este prezentată și în figura 2.1.3.1.

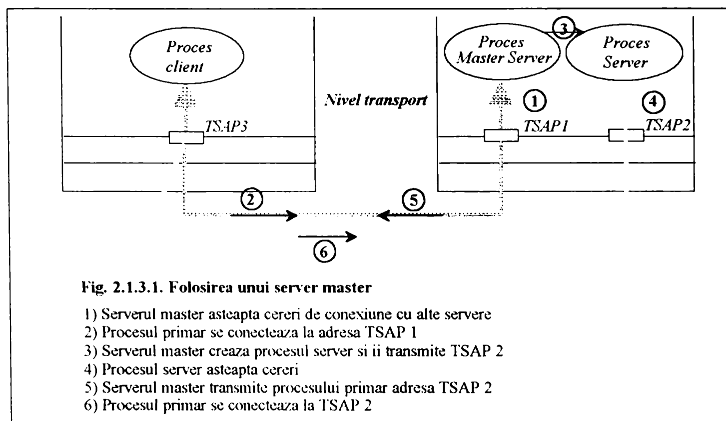


Fig. 2.1.3.1. Folosirea unui server master

1) Serverul master asteapta cereri de conexiune cu alte servere
2) Procesul primar se conecteaza la adresa TSAP 1
3) Serverul master creaza procesul server si ii transmite TSAP 2
4) Procesul server asteapta cereri
5) Serverul master transmite procesului primar adresa TSAP 2
6) Procesul primar se conecteaza la TSAP 2

3. O altă metodă de a furniza adrese TSAP este folosind un *server de nume (name server sau directory server)*, a cărui adresă TSAP este publică în rețea. Pentru a afla adresa TSAP corespunzătoare unui server, orice client stabilește întîi o conexiune cu serverul de nume, furnizează un nume generic global al serverului, primește adresa TSAP a acestuia și se deconectează de la serverul de nume, conectîndu-se în continuare la adresa TSAP determinată. Serverul de nume trebuie să mențină o tabelă cu corespondența nume server -adresă TSAP; orice server nou creat trebuie înregistrat de serverul de nume.

Din punctul de vedere al unui proiectant al nivelului transport, prezintă importanță calitatea serviciilor de rețea care îi pot fi oferite deoarece de aceasta va depinde complexitatea sa; cu cît serviciile de rețea care depind de tipul rețelelor și ale subrețelelor de comunicații sunt mai sigure, cu atît complexitatea nivelului de transport va fi mai mică. În acest sens, serviciile de rețea pot fi clasificate astfel: (A). Retele cu rata erorilor nesemnalate acceptabila și cu rata erorilor semnalate, de asemenea acceptabila (se considera că de acest tip de rețea se apropie rețelele locale LAN) [Tan88][Sta89]; (B). Rețele cu rata erorilor nesemnalate acceptabila dar cu rata erorilor semnalate inacceptabilă (în care se încadrează multe rețele WAN, de exemplu rețele publice X.25); (C). Rețele cu rata erorilor nesemnalate și semnalate inacceptabilă (rețele WAN care oferă servicii neorientate pe conexiune, interconectările de rețele). În acest context, o eroare este definită ca un pachet pierdut sau duplicat, o eroare semnalată este o eroare detectată și nerecuperată de nivelul rețea și, ca urmare semnalată nivelului transport iar o eroare nesemnalată este o eroare nedetectată de nivelul rețea si, ca urmare nerecuperată și neraportată. Chiar și in cazul rețelelor sigure, de tipul A, trebuie luate în considerare încă două aspecte care aduc complicații nivelului transport: limitarea dimensiunii unui pachet și secvențierea pachetelor (recepționarea pachetelor trebuie făcută în aceeași ordine în care au fost emise).

A. Gestiunea transferului datelor

Gestiunea transferului datelor la nivelul transport este evident influențată de tipul de rețea utilizat (A, B sau C) deoarece, pe de o parte, mecanismele de control al fluxului și de confirmare a pachetelor în rețele de tipul A sunt minimale pe cînd în rețelele de tipul C ele sunt complexe, iar, pe de altă parte, în rețelele de tipul C trebuie luate în considerare și erorile posibile în cursul transferului proceselor și deci construiți algoritmi pentru retransmisia pachetelor și verificarea corectitudinii lor.

S-ar putea spune că, în intenția de a corecta imperfectiunile nivelului rețea, nivelul transport va avea în ceea ce privește transferul datelor, funcții similare nivelului legaturii de date: detectarea erorilor, controlul fluxului; complexitatea soluțiilor adoptate la nivelul transport este însă mai mare, deoarece, dacă la nivelul legaturii de date două noduri comunică direct printr-un canal fizic, la nivelul transport două noduri comunică prin intermediul unei întregi subrețele, în care pachetele pot fi mai ușor pierdute, întîrziate și deci duplicate. Mecanismele pentru controlul fluxului și pentru rezervare de buffere sunt necesare în ambele nivele; datorită

BUPT

însă numărului mult mai mare de conexiuni existente la nivelul transport decît numărul de linii de la nivelul legaturii de date este ineficient ca la nivelul transport să se aloce individual, pentru fiecare conexiune, cîte un set de buffere, ca la nivelul legaturii de date. Strategia de rezervare a bufferelor este integrată în *mecanismul de control al fluxului*; acesta este mai complex decît controlul fluxului de la nivelul legăturii de date deoarece pe de o parte (a) implică interacţiunea a trei nivele, nivelul utilizator al nivelului transport, nivelul transport şi nivelul reţea, şi pe de altă parte, (b) timpul de transfer al datelor între două entităţi transport din noduri diferite este variabil, depinzînd de încărcarea reţelei. Figura 2.1.3.2. ilustrează aspectul (a) de mai sus: la transmiterea unui pachet de date de către procesul A (procesul A este utilizatorul nivelului de transport) către procesul B sunt implicate patru cozi de pachete de date. Astfel, procesul A generează pachetul şi îl depune în coada de pachete pentru a fi trimis către B; procesul A poate aştepta la operaţia send pînă: (1) are permisiunea nivelului transport din propriul nod (controlul fluxului la nivel de interfaţă) şi (2) are permisiunea nodului B (controlul fluxului capăt la capăt). După ce nivelul de transport a preluat pachetul, îl va memora într-o coadă pînă cînd are permisiunea de la nivelul reţea de a-l trimite, după care pachetul este trimis la nodul destinaţie de către nivelul reţea. La nodul destinaţie, nivelul reţea trebuie să păstreze în coadă pachetul pînă cînd are de la b permisiunea de a-l livra; analog b va memora pachetul pînă va avea permisiunea destinaţiei (procesul B) de a-i livra pachetul. Figura 2.1.3.3. ilustrează interacţiunile posibile, ca efect al întîrzierilor de mai sus. Se observă că există trei cazuri posibile de a confirma o operaţie *send*: (1) la acceptarea datelor pentru transfer de către nivelul de transport al emiţătorului (3 în fig.2.1.3.3.); (2) la acceptarea datelor de către nivelul transport al receptorului (6 în fig.2.1.3.3.); (3) la acceptarea datelor de către utilizatorul datelor din nodul destinatar (8 în fig. 2.1.3.3.).

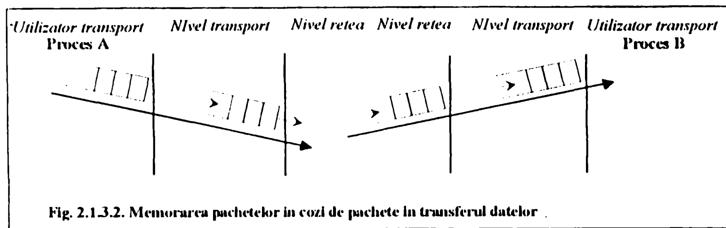


Fig. 2.1.3.2. Memorarea pachetelor în cozi de pachete în transferul datelor

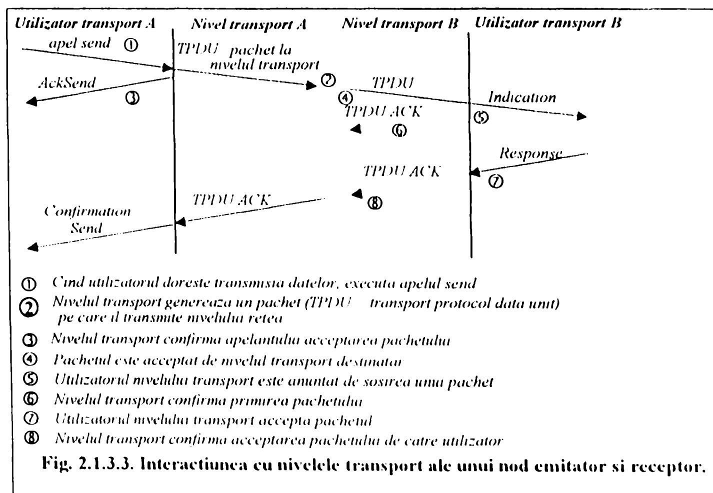


Fig. 2.1.3.3. Interactiunea cu nivelele transport ale unui nod emitator si receptor.

O concluzie a discuţiei de mai sus este faptul că există două motive pentru care nivelul transport al unui nod destinatar poate cere emiţătorului micşorarea ratei de transmisie a datelor: atunci cînd nivelul transport al destinaţiei nu poate face faţă fluxului de date de la emiţător sau cînd utilizatorul nivelului transport al destinaţiei nu poate face faţă fluxului de date. Ambele cauze se manifestă în ultima instanţă prin epuizarea capacităţii de memorare de la nivelul transport. În această situaţie, pentru controlul fluxului datelor emiţătorului se pot aplica următoarele metode: (1) receptorul nu execută nimic special; (2) receptorul refuză să primească alte pachete; (3) se utilizează un algoritm cu fereastra glisantă; (4) se utilizează o schema cu credite. Abordarea (1) este puţin eficientă; ea nu are ca efect micşorarea ratei de emisie a pachetelor de către emiţător, ci dimpotrivă, acesta nemaiprimind confirmările va trebui să retransmită, pe lîngă noile date şi

BUPT

datele neconfirmate încă. Metoda (2) transferă problema controlului fluxului la nivelul reţea: nivelul transport al receptorului refuză să mai primeasca date de la nivelul reţea şi acesta va trebui să rezolve controlul fluxului la nivelul său, nivelul reţea al emiţătorului refuzînd să mai primeasca date de la emiţător. Metoda (3) este cunoscută de la nivelul legaturii de date. Ea lucrează bine într-o reţea de tipul A, dar într-o reţea de tipul B sau C poate genera pachete duplicate nedorite. Metoda (4) este, în schimb, convenabilă în orice tip de reţea. Spre deosebire de (3) ea decuplează confirmarea pachetelor de problema controlului fluxului. Pe scurt, metoda constă tot într-un algoritm cu fereastra glisantă, dar, în cursul transferului, se transmite emiţătorului numărul de buffere ale receptorului ramase disponibile (evident în limita dimensiunii ferestrei); pentru a nu se genera noi pachete pentru credite, acestea se trimit în corpul confirmărilor. Prin urmare, pentru a confirma N pachete şi a acorda credite, un nivel transport receptor va trimite o confirmare de forma (ACK N+1, Credit M) ceea ce inseamnă confirmarea pachetelor 1...N şi acordarea de credite pentru pachetele N+1, ..., N+M (numai acestea pot fi transmise). Pentru a creşte sau descreşte creditul la X, fără ca alte pachete să fi fost recepţionate, receptorul va trimite o confirmare de forma (ACK N+1, Credit X); pentru a confirma un nou pachet primit, receptorul va trimite (ACK N+2, Credit M-1). În fig. 2.1.3.4. se prezintă un exemplu de transfer de date folosind acest algoritm. Chiar dacă algoritmul este utilizat într-o reţea de tipul C, el funcţionează corect şi eficient: dacă se pierde un pachet de confirmare, emiţătorul, în urma unei cereri de timeout va retransmite pachetul şi va primi confirmarea (şi creditul). Poate apare însă o interblocare cînd, după recepţionarea unei confirmări de tipul (ACK N, Credit 0), practic închiderea ferestrei, receptorul trimite emiţătorului (ACK N, Credit M) dar confirmarea este pierdută; emiţătorul asteaptă un nou credit de la receptor, care însă l-a trimis. Ca atare, pentru a inlatura interblocarea, trebuie setat un timer la trimiterea unui pachet de confirmare; cînd timpul a expirat, se repetă confirmarea (dacă se duplică cea precedentă nu este nici o eroare).

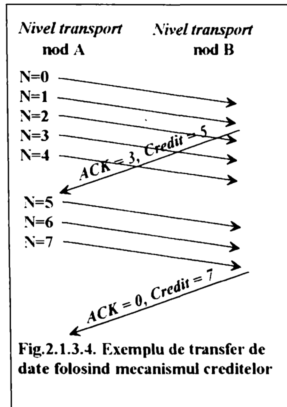


Fig.2.1.3.4. Exemplu de transfer de date folosind mecanismul creditelor

Pentru ambele metode de control al fluxului, la nivelul receptorului şi emiţătorului trebuie utilizată o metodă pentru *rezervarea bufferelor*, avînd în vedere că nivelul transport va avea de tratat mai multe conexiuni. Dacă se ţine cont de faptul că acestea se stabilesc şi desfiintează dinamic, atunci rezulta că este eficient ca *alocarea bufferelor* să se facă dinamic, la stabilirea conexiunior. Bufferele pot fi alocate separat, pentru fiecare conexiune sau, per global, pentru toate conexiunile dintre două noduri. Este eficient ca pentru conexiunile cu trafic mare (de exemplu transfer de fişiere) să se utilizeze buffere dedicate, iar pentru conexiunile cu trafic variabil (de exemplu cel produs de un terminal) să se rezerve bufferele pe masură ce sunt necesare. Un alt aspect legat de buffere este *dimensiunea lor* (fig.2.1.3.5.). Pot fi utilizate: (a) buffere de dimensiune fixa cu dezavantajul irosirii de spaţiu dar cu o gestionare simplă (b) buffere de dimensiune variabila, sau, pentru o mai bună utilizare a memoriei se poate aloca (c) un singur buffer, mare, circular pentru fiecare conexiune (metodă eficientă în cazul conexiunilor cu trafic mare).

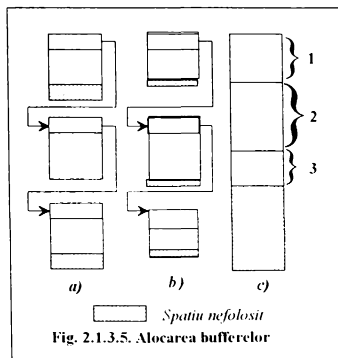


Fig. 2.1.3.5. Alocarea bufferelor

Alocarea dinamică a bufferelor determină decuplarea mecanismului confirmărilor de cel al controlului fluxului ceea ce implică utilizarea algoritmului cu credite. Pentru controlul fluxului datelor poate fi eficientă şi ajustarea ferestrei [Tan89]. Dimensiunea ferestrei depinde de capacitatea de transmisie a subreţelei: astfel, dacă subreţeaua poate transmite P pachete/sec şi durata ciclului unui pachet în reţea (incluzînd transmisia, memorarea în cozi, prelucrarea la nodul destinaţie şi returnarea confirmării) este C atunci fereastra emiţătorului ar trebui să fie P*C, pentru ca timpul ciclului unui pachet să fie folosit complet. Fereastra trebuie ajustată periodic deoarece capacitatea reţelei depinde de traficul pachetelor. În acest scop se pot masura parametrii P şi C: pentru C se alege o valoare medie, iar P se evaluează prin impărţirea numărului de confirmări primite într-o anumită perioadă de timp la acea perioada de timp (in timpul masuratorii, emiţătorul va trimite pachete la rata maximă de transmisie pentru a fi sigur că rata de primire a confirmărilor este limitată de capacitatea reţelei şi nu de rata de transmisie mai lentă).

Alte două aspecte ale transferului de date caracteristice tuturor tipurilor de servicii de reţea A, B, C sunt: (1) pastrarea ordinii de emisie a pachetelor şi la recepţie şi (2) fragmentarea pe pachete a unui mesaj primit de la nivelul superior. Aspectul (1) implică nu numai numerotarea în secvenţă a pachetelor din cadrul unei conexiuni, dar şi a pachetelor de confirmare, dacă se folosesc credite. Fig. 2.1.3.6. arată efectul

BUPT

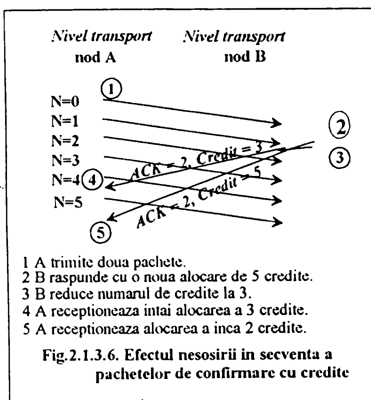


Fig.2.1.3.6. Efectul nesosirii in secventa a pachetelor de confirmare cu credite

dezastruos al recepționarii pachetului (ACK 2, Credit=3) inaintea pachetului (ACK 2, Credit=5).

Aspectul (2) trebuie luat în considerare pentru două cazuri distincte, avînd în vedere că aplicațiilor le sunt caracteristice două tipuri de transfer de date: (a) orientate pe flux, cînd utilizatorul nivelului transport acceptă datele în flux continuu, fără ca eventualele puncte de oprire să aibă semnificație și (b) orientate pe blocuri, cînd receptorul acceptă datele la nivel de bloc (un bloc este un mesaj) asa cum au fost transmise de emițător. În cazul b), dacă un bloc (TSDU în terminologia OSI) depășeste dimensiunea unui pachet, el trebuie fragmentat, și reasamblat la nodul recepție înainte de livrarea la utilizator. Pentru a se putea controla ordinea sosirii pachetelor unui bloc la destinație nu este nevoie de numerotarea pachetelor din interiorul blocurilor (aceasta ar conduce la o dubla numerotare deoarece există numărul de secvență al pachetelor dintr-o conexiune) ci este suficient ca ultimul pachet al unui bloc să fie marcat cu un indicator special de sfîrșit de bloc, EOT; toate pachetele următoare pînă la primul pachet cu indicatorul EOT inclusiv, vor fi considerate ca făcînd parte din următorul bloc s.a.m.d..

Verificarea corectitudinii datelor (obligatorie în cazul rețelelor din clasa C pentru a compensa eventualele erori nesemnalate ale pachetelor, în decursul parcurgerii subrețelei) se poate face prin adăugarea unei sume de control în headerul fiecărui pachet transmis. În cazul rețelelor de clasa C, implicații mari o are *strategia de retransmisie* a pachetelor neconfirmate. Aspectul critic în ceea ce privește strategia de retransmisie este stabilirea valorii timeout-ului; o valoare fixată nu raspunde cerințelor de adaptare la încărcarea rețelei; de aceea este indicată o valoare medie bazată pe masurarea întîrzierilor confirmărilor. Chiar și o schemă adaptivă pentru stabilirea timeout-ului gasește tot o valoare aproximativă deoarece destinația nu trimite confirmările imediat după primirea pachetelor (se folosesc algoritmi cu fereastră glisantă), încărcarea rețelei este variabilă iar măsurătorile pot fi imprecise pentru că dacă un pachet este retransmis nu se poate distinge între confirmarea pachetului initial și a celui duplicat. Un alt aspect critic legat de strategia de retransmitere este că secvența numerelor pachetelor dintr-o fereastra nu trebuie să se repete decît după cel puțin timpul maxim de viață al unui pachet în rețea. Fig. 2.1.3.7. ilustrează un contraexemplu pentru această cerință. Se presupune, pentru simplitate că se folosește pentru controlul fluxului un algoritm cu fereastră glisantă de dimensiune 3 (deci un spațiu de 8 numere). Se observa că dacă pachetul 0 apare la destinație după epuizarea unei secvente complete, deregleaza intregul flux al datelor.

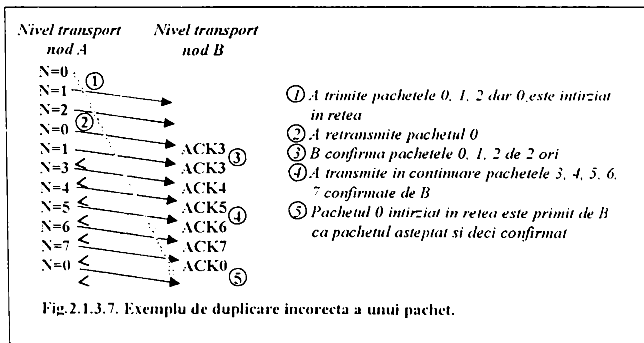


Fig.2.1.3.7. Exemplu de duplicare incorecta a unui pachet.

B. Stabilirea și desființarea conexiunilor

În scopul implementarii serviciilor orientate pe conexiuni sunt necesare, pentru toate tipurile de rețele, A, B, C proceduri pentru stabilirea, respectiv desființarea unei conexiuni. Stabilirea unei conexiuni serveste, in principal, pentru 3 scopuri: (1) permite fiecărui capăt să se asigure că celălalt există; (2) permite negocierea unor parametri opționali (dimensiunea pachetelor, dimensiunea ferestrei, calitatea serviciului) (3) declanșează alocarea resurselor necesare conexiunilor (buffere, intrări în tabela de conexiuni).

Deoarece la nivelul transport pot exista mai multe conexiuni deschise pe aceeași masină, fiecărei conexiuni trebuie să i se asocieze un număr care se depune în fiecare pachet, în scopul recunoașterii apartenenței pachetelor de către receptor. Conexiunile se stabilesc prin acord reciproc, in urma unor comenzi de la utilizatorul nivelului transport și a unor pachete de control, așa cum se prezintă în diagrama din fig. 2.1.3.8. Se observă că utilizatorul semnalează că așteaptă o cerere de conexiune printr-o comandă *Listen*, care declanșează tranziția IDLE -> LISTEN (un exemplu ar fi un server de fișiere); utilizatorul poate închide

BUPT

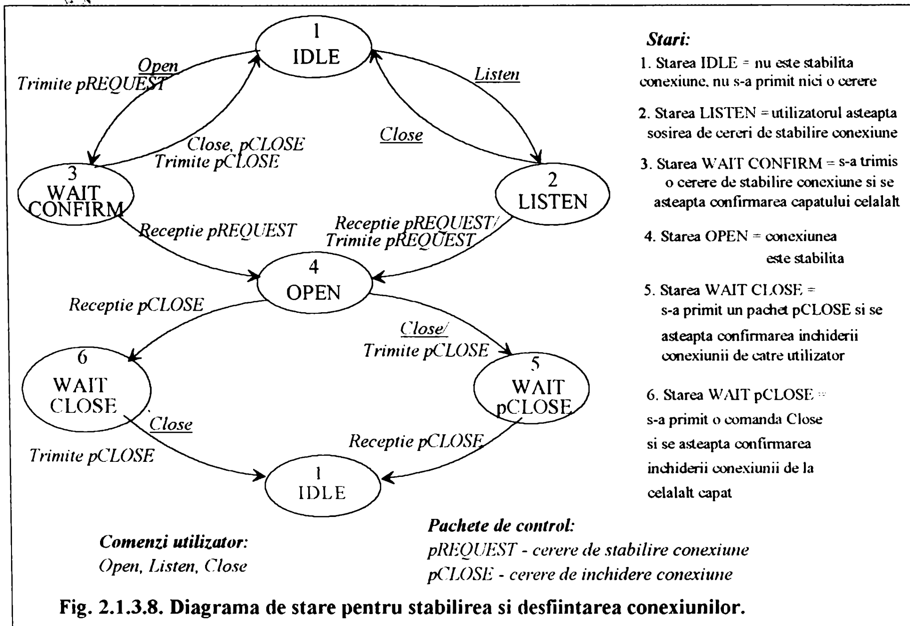
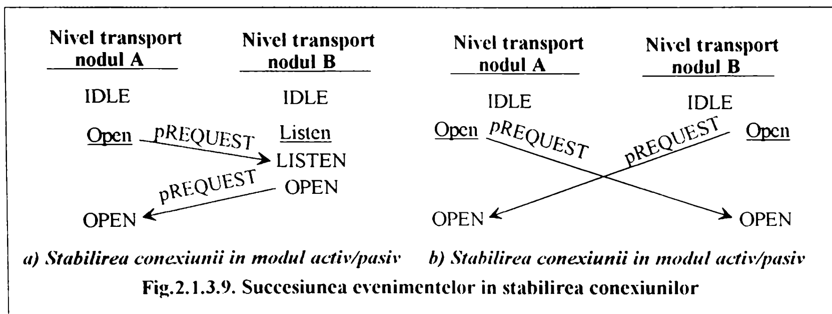


Fig. 2.1.3.8. Diagrama de stare pentru stabilirea si desfiintarea conexiunilor.

conexiunea trimitînd o comanda *Close*. Daca în starea LISTEN se primeste o cerere de stabilire conexiune pREQUEST (pachet de control *request for connection*) care specifică ca destinație utilizatorul aflat în așteptare, atunci se stabileste o conexiune. În acest caz, nivelul transport trimite o confirmare pREQUEST nivelului transport pereche și transferă conexiunea în starea OPEN. Un utilizator poate deschide o conexiune executînd o comandă *Open*, care declanșează trimiterea unui pachet pREQUEST nivelului transport pereche. Recepția unui pachet pREQUEST de la nodul pereche stabilește o conexiune. Conexiunea poate fi desființată prematur dacă utilizatorul local execută o comandă *Close* sau nivelul transport pereche refuză conexiunea trimitînd un pachet pCLOSE. Figura 2.1.3.9. arată succesiunea evenimentelor în stabilirea conexiunilor. Se observă că stabilirea conexiunii se face corect chiar și în cazul cînd ambele capete încearca să o stabilească aproximativ în același timp.



a) Stabilirea conexiunii in modul activ/pasiv *b) Stabilirea conexiunii in modul activ/pasiv*

Fig.2.1.3.9. Succesiunea evenimentelor in stabilirea conexiunilor

În cazul în care se recepționează un pachet pREQUEST cînd utilizatorul nu a cerut așteptarea cererilor de stabilire a conexiunii există trei soluții: (1) nivelul transport poate rejecta cererea trimițînd un pachet pCLOSE (2) cererea este memorată într-o coada de cereri pînă utilizatorul executa o comandă *Listen* (3) nivelul transport poate intrerupe sau semnaliza într-un mod oarecare utilizatorului că există o cerere de stabilire conexiune în așteptare. În ultimul caz, nu mai este strict necesară o comanda Listen; ea poate fi înlocuită printr-o comandă Accept, care este vazută ca un semnal de la utilizator către nivelul transport că accepta cererea de conexiune (aceasta este soluția adoptată de ISO).

BUPT

Desființarea conexiunilor se face similar; desființarea poate fi cerută de orice capăt, de ambele capete sau de nivelul transport însuși la ambele capete. Conexiunea se desființează prin acord reciproc; strategia permite o desființare bruscă sau lentă. În ultimul caz, o conexiune în starea 5 (se așteaptă pCLOSE) trebuie să continue să accepte date pînă la recepția unui pachet pCLOSE.

Diagrama prezentată în fig.2.1.3.8. este valabilă pentru rețele de tipul A. Pentru celelalte tipuri de rețele trebuie luate în considerare aspecte legate de (1) nesosirea în secvență a pachetelor (2) generarea de către nivelul rețea a unei primitive N-RESET pentru a semnala o situație catastrofală în funcționare (rețea supraîncărcată, probleme hardware, software) (3) erori în transmisie și apariția de pachete duplicate.

Situațiile de tipul (1) pot provoca recepția unui (sau mai multor) pachete de date înainte de recepția acceptarii conexiunii trimisă de capatul pereche; în acest caz este necesar să se depună aceste pachete într-o coadă de pachete pînă la recepția acceptarii conexiunii. O situație asemănătoare poate apare și la desființarea conexiunilor: după ce un nivel transport a trimis ultimul pachet de date, trimite pachetul de control pCLOSE, care poate ajunge la destinație înaintea ultimului pachet de date. O soluție pentru a evita astfel de probleme este ca pachetul pCLOSE să conțină numărul de secvență de după ultimul pachet transmis; astfel, nivelul transport destinație va aștepta eventualele pachete întîrziate și după ce a recepționat pCLOSE, fără închiderea conexiunii.

Se consideră că situațiile de tipul (2) pot apare foarte rar în rețele de tip A; în cazul apariției lor se închid conexiunile fără a se mai încerca recuperarea lor. Pentru casele B, C se încearcă o resincronizare după un N-RESET în vederea determinării de către fiecare capăt a pachetelor trimise, și recepționate (ceea ce înseamnă că trebuie reținute copii ale pachetelor trimise, dar neconfirmate, pentru a putea fi retransmise după un N-RESET) și stabilirea unei noi conexiuni la nivelul rețea.

Situațiile de tipul (3) pot genera probleme la stabilirea conexiunilor. Iată un exemplu: dacă după închiderea unei conexiuni, mai există pachete duplicate și dacă se deschide imediat o altă conexiune între aceleași entități. Se poate întîmpla ca pachetele duplicate, întîrziate aparținînd primei conexiuni să fie acceptate ca valabile și în cadrul celei de-a doua conexiuni; analog și confirmările întîrziate pot fi acceptate în alte conexiuni și crea probleme. Exista mai multe soluții pentru aceste situații: (a) extinderea numerelor de secvență ale pachetelor și după închiderea conexiunilor. Aceasta inseamnă că nivelul transport trebuie să cunoască numerele ultimelor pachete ale conexiunilor închise, astfel încît la stabilirea unei noi conexiuni să aleagă un alt număr de secvență pentru primul pachet pe care îl transmite capătului celalalt în pREQUEST. Acest procedeu este simetric; fiecare capăt trebuie să-și declare la stabilirea conexiunii numărul de secvență al primului pachet; ele nu trebuie să fie egale, pentru cele două capete; (b) utilizarea unui identificator de conexiune separat pentru fiecare tranzacție, ales de capătul ce initiaza conexiunea (de exemplu, la stabilirea unei conexiuni se incrementează identificatorul) care se depune în pachete; la închiderea fiecărei conexiuni, nivelul transport actualizează o tabela a conexiunilor terminate (fiecare intrare conține identitatea celuilalt capăt și identificatorul conexiunii). Ambele procedee de mai sus se comportă bine numai dacă nu apare o avarie a nodului, cînd se pierd atît numerele de secvență ale ultimelor pachete ale conexiunilor închise cît și identificatorii conexiunilor închise. În acest caz, singura soluție este limitarea duratei de viață a pachetelor în rețea și deci, după avaria unui nod se așteaptă timpul maxim de viata al pachetelor după care se pot deschide noi conexiuni fără probleme. Pentru limitarea duratei de viata a pachetelor se folosesc 3 metode: (a) limitarea circulației pachetelor în subrețea (se previne astfel circulatia în bucla închisă a pachetelor) (b) folosirea unui contor al nodurilor intermediare parcurse de un pachet, care se depune în fiecare pachet; acesta se initializează

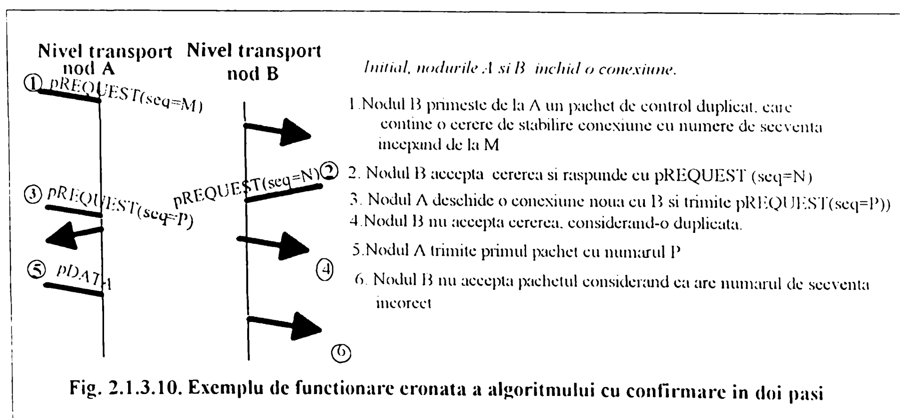


Fig. 2.1.3.10. Exemplu de functionare eronata a algoritmului cu confirmare in doi pasi

BUPT

cu o valoare maximă permisă şi se decrementează, în fiecare nod intermediar. Cînd contorul ajunge la zero, protocolul legaturii de date refuză să primeasca pachetul. (c) folosirea unei marci de timp pentru fiecare pachet. La crearea unui pachet se inițializează marca cu valoarea timpului curent, urmînd ca pachetul să fie ignorat cînd devine mai vechi decît o anumită perioadă de timp stabilita. Metoda este însă mai dificil de implementat deoarece necesită ca ceasurile nodurilor să fie sincronizate.

Pachetele duplicate pot avea însă şi alte efecte dezastruoase la stabilirea conexiunilor. Figura 2.1.3.10. arată o secvență de pachete de control în urma căreia algoritmul de stabilire a conexiunilor prezentat, şi numit ***algoritmul cu confirmare în 2 paşi (two-way handshake)*** nu funcționează corect dacă ignoră duplicatele cererilor de stabilire conexiuni după stabilirea conexiunilor. Eroarea poate fi corectată dacă fiecare capăt confirma cererea de stabilire şi numărul de secvență al celuilalt. Algoritmul este cunoscut sub numele de ***algoritm cu confirmare în 3 paşi***; el conduce la modificarea diagramei de stari la cea prezentată în figura 2.3.1.11.

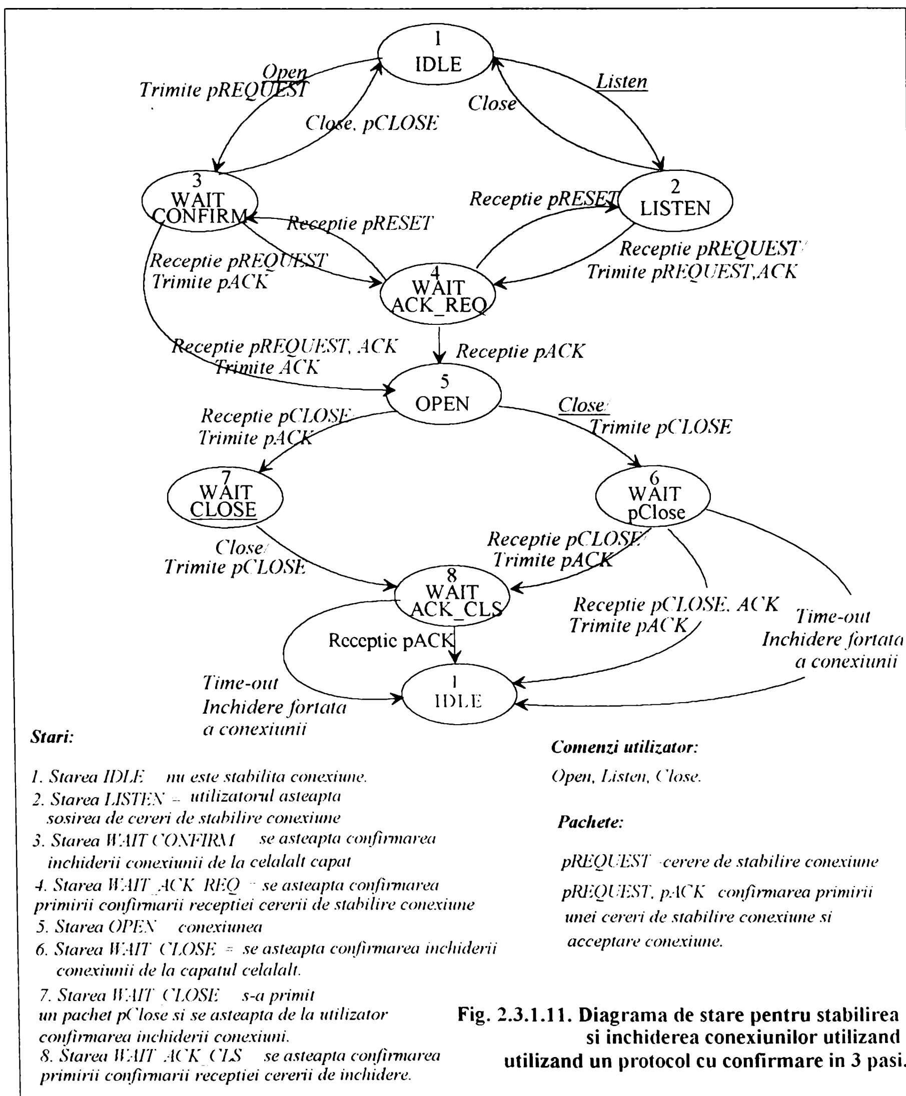


Fig. 2.3.1.11. Diagrama de stare pentru stabilirea si inchiderea conexiunilor utilizand utilizand un protocol cu confirmare in 3 pasi.

BUPT

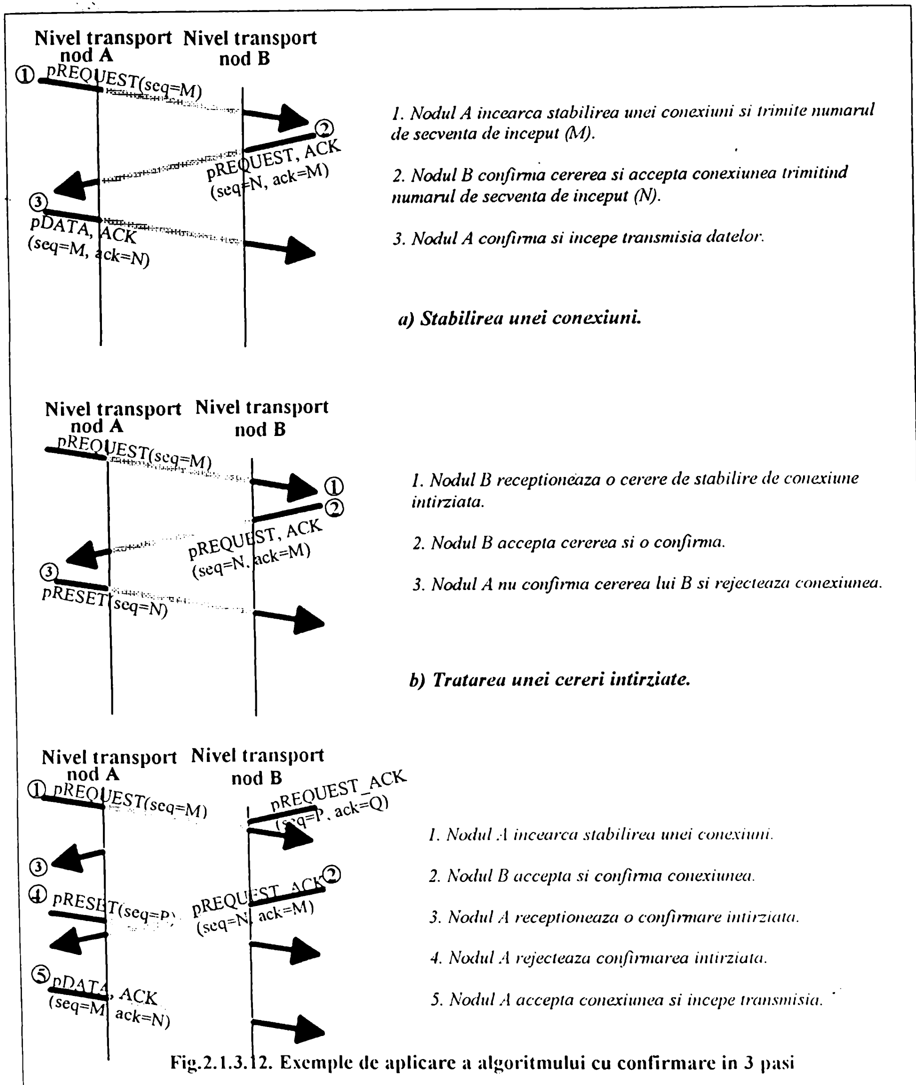


Fig.2.1.3.12. Exemple de aplicare a algoritmului cu confirmare in 3 pasi

Față de diagrama din fig. 2.1.3.8, apare o stare în plus pînă la stabilirea conexiunii, starea WAIT_ACK_REQ și un tip de pachet de control nou, pachetul pRESET pentru a reseta capătul celălalt cînd s-a detectat recepția unei cereri de stabilire pREQUEST duplicate.

Figura 2.1.3.12 prezintă aplicarea algoritmului de confirmare în trei pași în trei cazuri: se observă că în cazurile b) și c) de apariție a unui pachet duplicat a unei cereri de stabilire conexiune, respectiv a unui pachet duplicat a confirmării unei cereri, algoritmul lucrează corect.

Pentru simplificarea diagramei de stare din fig. 2.1.3.11, nu au fost reprezentate tranzițiile în care se transmite pRESET. Pentru acestea, regula de bază este: se transmite pRESET dacă starea conexiunii nu este OPEN și dacă se primește o confirmare ACK invalidă (care refera o cerere ce nu s-a trimis).

Pentru închiderea unei conexiuni se poate folosi o soluție asemanatoare: fiecare capăt trebuie să confirme explicit fiecare pachet pCLOSE al celuilalt. Numerele de secvență se folosesc astfel: (a) pachetul pCLOSE conține numărul de secvență plus unu al ultimului pachet de date trimis; (b) pachetul ACK conține numărul de secvență primit în pachetul pCLOSE. Pentru a elimina interblocarea care poate apare în cazul

BUPT

pierderii de pachete, trebuie folosite mecanisme de timeout (fig.2.1.3.11); după un număr oarecare de retransmisii, conexiunea este închisă forţat.

În cazul avariei unui nod (si repunerii în funcţiune apoi a lui) toate informaţiile despre conexiuni vor fi distruse, iar conexiunile afectate devin "deschise pe jumătate". Capătul activ al conexiunii poate închide conexiunea cu ajutorul unui timer care masoară timpul total de aşteptare a unei confirmari după un număr maxim de retransmisii; dacă se depăşeste acest timp se închide conexiunea şi se semnalează utilizatorului nivelului transport închiderea forţată. În cazul refacerii imediate a nodului cazut, închiderea conexiunii se poate face şi altfel: nivelul transport trebuie să returneze RESET X la orice pachet; capătul activ al conexiunii, va verifica validitatea numărului de secvenţă X (pentru a depista dacă RESET este raspunsul pentru un pachet întîrziat); în cazul unui RESET activ, nivelul transport va declanşa închiderea forţată a conexiunii. Resincronizarea între cele două noduri nu poate fi făcută însă la nivelul transport, deoarece nu se poate determina cîte pachete au fost transmise corect.

2.1.4. Nivele suport pentru aplicaţii

Nivelele superioare ale modelului ISO/OSI - nivelul *sesiune, prezentare* şi *de aplicaţii* - realizate pe baza serviciilor de comunicaţie independente de reţea furnizate de nivelul transport au sarcina de a furniza unui proces utilizator facilităţile necesare schimbului de informaţii în cadrul unui mediu de prelucrare distribuită a informaţiei. Ele sunt referite colectiv ca *nivele suport pentru aplicaţii* sau *nivele specifice aplicaţiilor.*

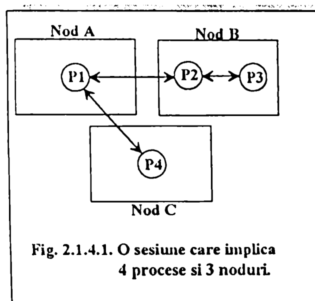


Fig. 2.1.4.1. O sesiune care implica 4 procese si 3 noduri.

Nivelul sesiune extinde comunicaţia dintre două procese, prezentată la nivelul transport; astfel, la acest nivel, comunicaţia între procese apare în contextul unei aplicaţii, divizată în subtaskuri, fiecare task putînd fi mapat în unul sau mai multe procese. Nivelul sesiune gestionează comunicaţia între o pereche de astfel de procese (fig.2.1.4.1) oferind suport pentru stabilirea şi terminarea sesiunilor de transmisii, între procese şi pentru organizarea şi sincronizarea dialogului dintre ele. Serviciile acestui nivel, definite de ISO 7498, sunt descrise în fig. 2.1.4.2.

	Servicii	*Descriere*
(1)	Stabilirea unei conexiuni sesiune	Folosita pentru conectarea entitatilor prezentare
(2)	Eliberarea unei conexiuni sesiune	Eliberarea se poate face după transmiterea în ambele direcții a unitatilor de date (SSDU) în aşteptare
(3)	Serviciu pentru date în carantina	Permite entitaţii prezentare de a cere carantina (bufferare) pentru una sau mai multe unitati de date (SSDU)
(4)	Transferul datelor normale şi a datelor expeditive	Transfer de date
(5)	Gestiunea interactiunilor	Dialogul intre 2 procese poate fi în ambele directii simultan, în ambele directii alternind sau intr-o singură directie
(6)	Sincronizare	Permite entitatilor prezentare definirea de puncte de sincronizare şi resincronizare în raport cu aceste puncte
(7)	Raportarea excepţiilor	Anunta aparitia unei erori entitaţii prezentare

Fig. 2.1.4.2. Serviciile nivelului sesiune

Se apreciază [Tan88] că nivelul sesiune este implementat mai puţin în diverse arhitecturi de reţele, fiind o invenţie OSI.

Nivelul prezentare are drept focar de interes sintaxa datelor schimbate între entităţile aplicaţie; scopul său este de a rezolva diferentele în ceea ce priveşte formatul şi reprezentarea datelor. Acest lucru este necesar deoarece la nivelul aplicaţie, prezintă importanţă *semantică datelor*, iar de la nivelul sesiune pînă la nivelul fizic datele sunt vazute ca şiruri de octeţi. Este sarcina nivelului prezentare de a furniza reprezentarea datelor de la nivelul aplicaţie într-o forma binară, cu aceeaşi semnificaţie, la ambele capete, de aceea centrul său de interes este *sintaxa datelor* de la nivelul aplicaţie. Deoarece este practic imposibil de a separa complet semantica datelor de sintaxa lor, ISO a propus următoarea soluţie: (fig.2.1.4.3.): (1) la nivelul aplicaţie informaţia este reprezentată într-o *sintaxă abstractă*, care specifică datele la nivel de tipuri şi valori de date,

BUPT

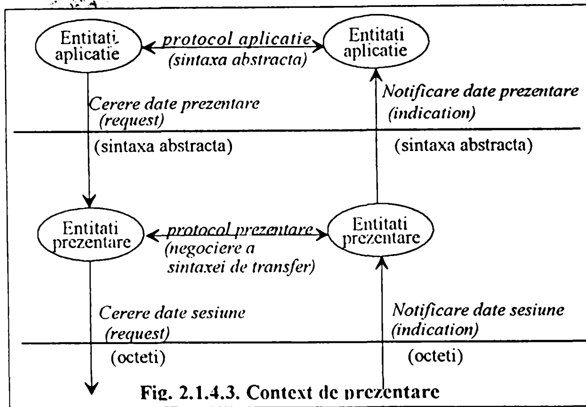


Fig. 2.1.4.3. Context de prezentare

independent de orice reprezentare specifică (protocoalele la nivel de aplicaţie vor descrie unităţile lor de date în termenii sintaxei abstracte) (2) nivelul prezentare comunică cu nivelul aplicaţie în termenii sintaxei abstracte (3) nivelul prezentare translatează datele reprezentate în sintaxa abstracta într-o *sintaxa de transfer*, sub o forma binara, utilizată pentru interacţiunea cu nivelul sesiune. *Sintaxa de transfer* defineşte reprezentarea datelor schimbate între entităţile prezentare; translatarea sintaxa abstracta -> sintaxa de transfer se face cu ajutorul unor reguli de codificare, care specifică reprezentarea valorii fiecărui tip de date. Fiecare entitate prezentare recunoaşte sintaxa abstractă a utilizatorului său şi una sau mai multe sintaxe de transfer disponibile pentru codificare (de exemplu, pentru o aceeaşi sintaxa abstracta pot exista mai multe variante ale unei sintaxe de transfer: o variantă care realizează codificarea datelor, alta care realizează criptarea, alta care realizează şi criptare şi codificare s.a.m.d); alegerea unei sintaxe de transfer se realizează prin negociere între două entităţi prezentare. Combinaţia dintre o sintaxa abstractă şi o sintaxa de transfer este denumita *context de prezentare*. ISO reuneşte definiţiile pentru o sintaxă abstractă (ASN.1) în standardul DIS 8824, iar regulile de codificare în standardul DIS 8825 (Basic Encoding Rules).

În arhitectura OSI funcţiile nivelului prezentare pot fi rezumate astfel: (1) negociere pentru acceptarea unei sintaxe de transfer convenabila pentru reprezentarea datelor la ambele capete (2) conversia datelor la acest format la ambele capete (3) maparea cererilor de servicii ale aplicaţiilor în funcţii, precum controlul dialogului şi al sincronizarii în primitivele corespodente ale serviciului sesiune. Deoarece inainte de a fi transmise nivelului sesiune, datele pot fi criptate şi comprimate, criptarea şi comprimarea datelor sunt alte aspecte legate de nivelul prezentare.

Nivelul aplicaţie ca nivel de graniţă între mediul OSI şi procesele utilizator de aplicaţie, furnizează servicii acestor procese (aflate deci în mediul exterior arhitecturii OSI) în contextul unei prelucrari distribuite. Nivelul aplicaţie este compus din cateva module separate, denumite *elemente de servicii*. Unele dintre aceste elemente sunt comune multor aplicaţii şi de aceea referite ca *elemente de servicii comune aplicaţiilor* (CASE= Common Application Service Elements); ele includ funcţii precum gestiunea conexiunilor aplicaţie (modulul ACSE=Association Control Service Element) şi coordonarea acţiunilor entităţilor aplicaţie (modulul CCR= Commitment Concurrency and Recovery). Alte elemente de servicii au fost prevazute special pentru a furniza servicii specifice şi de aceea sunt referite ca *elemente de servicii ale unor aplicaţii specifice* (Specific Application Service Elements); ele includ servicii precum: (1) accesul dintr-un proces utilizator la un sistem de fişiere aflat la distanţă - modulul FTAM (File Transfer Acces and Management) - (2) comunicarea a două procese utilizator care gestionează dialoguri cu terminale locale într-o manieră standardizată, independentă de caracteristicile şi provenienţa terminalului - modulul VT (Virtual Terminal) - (3) furnizarea unei facilităţi generale de poştă electronică - modulul MHS (Message Handling Service) - (4) obţinerea adresei de reţea a utilizatorilor pornind de la numele lor simbolice - modulul DS (Directory Services). Structura generală a nivelului aplicaţie este reprezentată în figura 2.1.4.4.

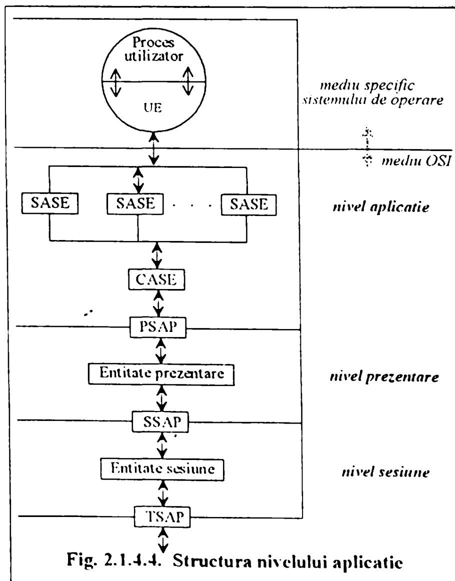


Fig. 2.1.4.4. Structura nivelului aplicatie

BUPT

Interfaţa dintre serviciile furnizate de nivelul aplicaţie şi procesele utilizator, denumită în general *elementul utilizator (UE= User Element)* sau *agentul utilizator (UA= User Agent)* sau *interfaţă utilizator (UI= User Interface)* se implementează printr-un set de funcţii sau proceduri de bilbiotecă, linkeditate cu procesele utilizator. Primitivele interfeţei dintre procesele utilizator şi UE nu trebuie să fie aceleaşi primitive (standard) furnizate de un element serviciu al entităţii aplicaţie. Neexistînd aceasta restricţie, pot fi folosite primitive specifice sistemului de operare local; este în sarcina UE-ului să realizeze corepondenţa necesară. Acesta a fost de altfel unul din scopurile ISO pentru nivelul aplicaţie: decuplarea mediului OSI de mediul specific sistemului de operare. Decuplarea se realizează prin următoarea manieră generală, comună tuturor elementelor de servicii ale unor aplicaţii specifice (fig.2.1.4.5): în primul rînd, pentru fiecare element se defineşte un dispozitiv virtual, împreuna cu un set de primitive standard pentru utilizarea lui. Toate cererile şi raspunsurile de la nivelul mediului OSI se fac folosind primitivele standard. Pentru maparea primitivelor relative la dispozitivul real (cereri şi raspunsuri), de la nivelul proceselor utilizator în primitive standard se foloseşte UE.

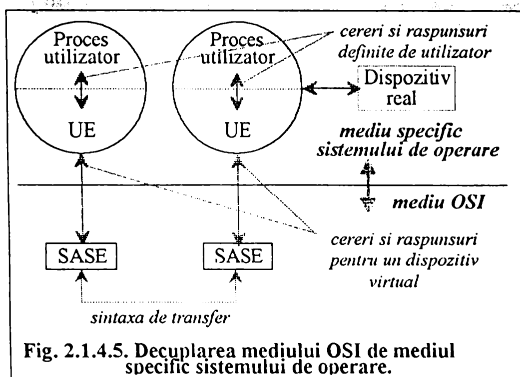


Fig. 2.1.4.5. Decuplarea mediului OSI de mediul specific sistemului de operare.

2.1.5. Solutii de implementare a modelului OSI

Deşi modelul OSI prevede impărţirea funcţiilor pe şapte nivele, numărul de nivele într-o implementare efectivă în sistemul de operare ca şi diviziunea funcţiilor pe nivele poate diferi într-o anumită masură. Se prezintă în acest subcapitol două implementări posibile ale modelului OSI, conceptual diferite.

Prima implementare se prezintă în figura 2.1.5.1 (se face abstracţie de gestiunea memoriei). Un proces poate comunica fie cu un proces local, fie cu unul de la distanţă. Interfaţa 1 transmite un apel fie modulului de comunicaţii interprocese, fie modulului corespunzator nivelului sesiune, în funcţie de localizarea procesului destinaţie: local sau la distanţă; interfaţa 2 furnizează servicii de I/O.

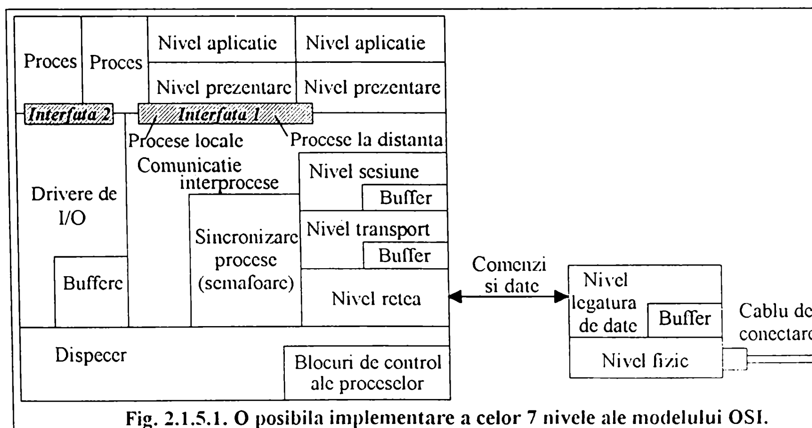


Fig. 2.1.5.1. O posibila implementare a celor 7 nivele ale modelului OSI.

A doua implementare utilizează un procesor separat pentru funcţiile de comunicaţie, considerînd că timpul necesar pentru acestea este considerabil si, ca urmare, reduce performantele sistemului în raport cu aplicaţiile. Arhitectura generală a unui astfel de sistem este prezentată în figura 2.1.5.2; subsistemul de comunicaţie este implementat pe o placa separată a calculatorului gazda. Aceasta conţine în afară de procesorul rezervat pentru comunicaţie, o memorie locală, necesara pentru protocoalele software (informaţii de stare), o memorie partajată, necesară pentru bufferele de mesaje (care trebuie transmise de la un nivel la

BUPT

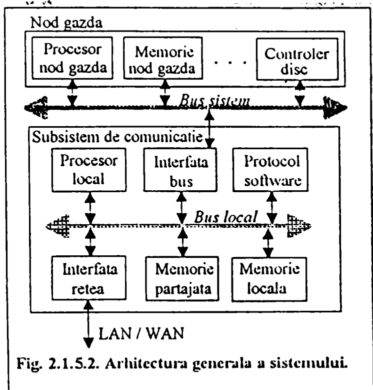


Fig. 2.1.5.2. Arhitectura generala a sistemului.

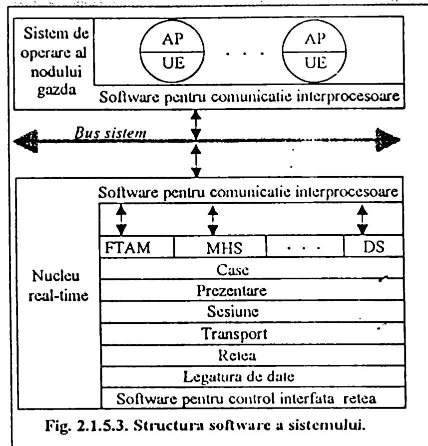


Fig. 2.1.5.3. Structura software a sistemului.

altul) şi circuite de interfaţare cu reţeaua de comunicaţii. La memoria partajată trebuie să aibă acces interfaţa cu reţeaua, procesorul local şi procesorul nodului gazdă; în acest mod toate bufferele pot fi accesate direct de fiecare, astfel încît între nivele nu se mai transmit mesaje, ci se transmit direct pointeri la mesaje. Structura software-ului acestei arhitecturi este prezentată în figura 2.1.5.3; subsistemul de comunicaţie este implementat sub forma de procese separate: cîte un proces pentru fiecare entitate de protocol plus un proces pentru gestiunea lor şi funcţii de timer. Procesele pot comunica între ele prin liste FIFO sau cutii poştale.

Comunicarea interprocese este implementată de un nucleu local real-time, responsabil şi pentru planificarea la execuţie a proceselor şi tratarea întreruperilor ca şi pentru gestionarea transferului de date la/de la interfaţa cu reţeaua. Comunicarea între subsistemele de comunicaţie şi nodul gazdă se face printr-un modul software de comunicaţie interprocesor, pentru care există cîte o copie în fiecare procesor. Pentru a se asigura sincronizarea transferului de informaţie între cele două procesoare, se folosesc întreruperi astfel: cînd procesorul nodului gazda doreşte, de exemplu, să transmită informaţii, scrie mai întîi informaţiile în bufferul asociat entităţii aplicaţie respective şi generează apoi o intrerupere pentru procesorul subsistemului de comunicaţie. Rutina de tratare a întreruperii determină entitatea SASE implicată, căreia îi transmite în coada de intrare pointerul la mesajul transmis.

2.1.6. Importanţa modelului OSI pentru sisteme distribuite.

Ansamblul de protocoale OSI a fost creat în scopul interconectarii sistemelor deschise; ele permit comunicarea între noduri fără a impune o faza de negociere a protocoalelor folosite de fiecare nod (acest lucru a fost scopul pe care şi l-a propus ISO).

Intenţia ISO/OSI nu a fost de a furniza standarde pentru sistemele distribuite. Se apreciază ([Gos91][Mul89]) că standardele internationale conţin prea multe prevederi pentru a oferi o comunicaţie eficientă pentru sistemele de operare distribuite (vezi sectiunea 2.2); pe de altă parte acestea pot să nu fie necesare datorită următoarelor motive: (1) sistemele de operare distribuite sunt în general proiectate de o echipă de specialişti, lucrînd în stransă colaborare (2) pe fiecare maşină se execută o copie a nucleului aceluiaşi sistem de operare (3) un proces client care trimite o cerere cunoaşte exact formatul şi structura cererilor acceptate de server, ca şi al raspunsurilor de la server.

Totusi, se apreciază ([Mul89]) că se impune o considerabilă presiune, atît din partea unor organizaţii europene cît şi din motive politice, asupra proiectantilor de sisteme distribuite de a utiliza protocoale ISO/OSI. Pe de altă parte, efortul de standardizare pe plan international este continuat pentru a produce standarde pentru prelucrari distribuite deschise (Open Distributed Processing); efortul a fost susţinut de ISO (International Standards Organization), ECMA (European Computer Manufacturers Association) şi ANSA (Advanced Networked Systems Architecture Project).

BUPT

2.2. Exemple de protocoale

2.2.1. Familia de protocoale TCP/IP

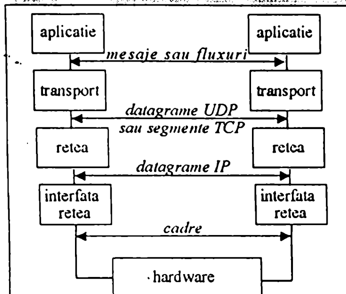


Fig. 2.2.1.2. Nivelele protocoalelor TCP/IP.

Deoarece familia de protocoale TCP/IP, este, la ora actuală utilizată pe scară largă, începînd de la calculatoare personale şi terminînd cu supercomputere atît în LAN-uri cît şi în WAN-uri, dar şi în sisteme de operare distribuite experimentale ca suport pentru primitivele de comunicare interprocese, se prezintă în aceasta secţiune elementele caracteristice, insistîndu-se pe protocolul IP versiunea 6 considerat ca o noua generaţie de protocol interreţele.

Familia de protocoale TCP/IP şi raportarea acestora la arhitectura ISO/OSI, este prezentată în figura 2.2.1.1, iar în figura 2.2.1.2 se prezintă structurarea protocoalelor pe nivele.

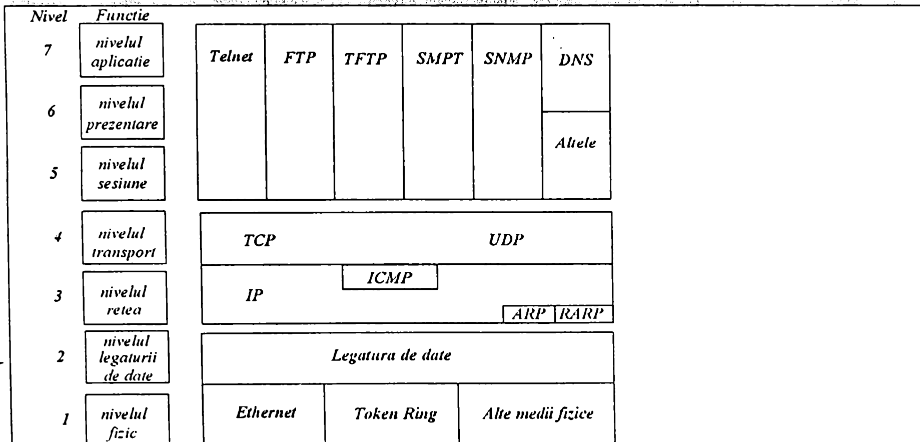


Fig. 2.2.1.1. Raportul OSI - TCP/IP.

ARP - Address Resolution Protocol - Mapeaza adresele IP în adrese fizice.

RARP - Reverse Address Resolution Protocol - Mapeaza adresele fizice în adrese IP.

IP - Internet Protocol - furnizeaza serviciul de livrare al pachetelor pentru TCP, UDP, ICMP.

ICMP- Internet Control Message Protocol - controleaza transmisia mesajelor de eroare şi control intrenoduri s porti. Mesajele (datagrame IP) sunt generate de software-ul de reţea, nu de procesele utilizator.

UDP - User Datagram Protocol - protocol neorientat pe conexiune, care nu garanteaza livrarea sigură a datagramelor.

TCP - Transmission Control Protocol - protocol orientat pe conexiune, garanteaza o transmisie sigură full-duplex, sub forma de flux de octeti.

TELNET -Protocol pentru emularea unui terminal

FTP - File Transfer Protocol - protocol pentru transferul fişierelor între două noduri; ofera un set bogat de facilităţi pentru autentificare, conversie date, parcurgere directoare.

TFTP - Trivial File Transfer Protocol - protocol pentru transferul fişierelor între două noduri; nu ofera toate facilităţile protocolului FTP.

SMTP - Simple Mail Transfer Protocol - protocol pentru posta electronica; furnizeaza specificaţii pentru interacţiuni şi formatul mesajelor de control.

SNMP - Simple Network Management Protocol - protocol pentru gestiunea reţelei; permite schimbul de informaţii între noduri relativ la configuraţie şi stare. Informatia disponibila este definita de un set de obiecte gestionate (statistici relativ la trafic, starea reţelelor conectate, contor al erorilor întîlnite; conţinutul tabelelor de rutare IP). Standardul MIB (Management Information Base) defineşte numele şi semantică variabilelor pe care serverele SNMP le menţin.

DNS - Domain Name System - Un server DNS translateaza numele simbolice în adresa IP.

BUPT

A. Protocoale IP. Noua generaţie de protocol IP, versiunea 6.

Protocolul IP versiunea 4, aflat în exploatare, în momentul curent, foloseşte pachete de maxim 64Ko; header-ul unui pachet, format dintr-o parte fixă de 20 octeţi şi o parte opţională de lungime variabilă, este prezentat în fig.2.2.1.3.

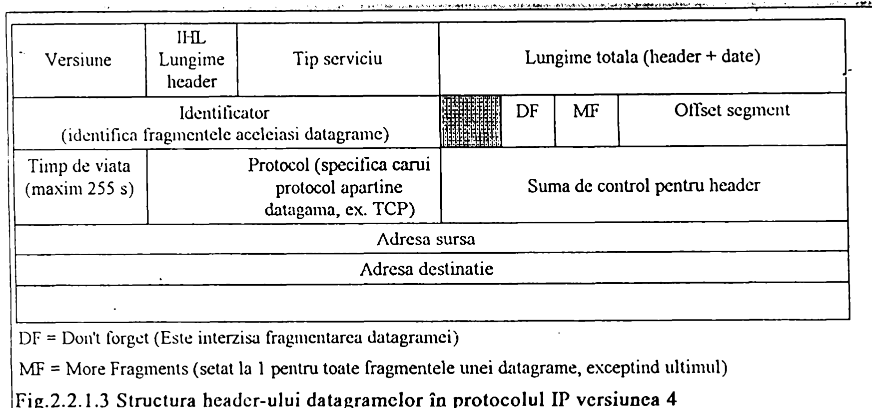


Fig.2.2.1.3 Structura header-ului datagramelor în protocolul IP versiunea 4

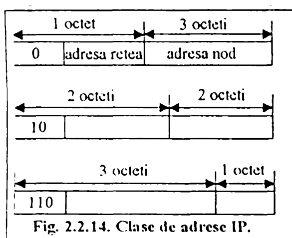


Fig. 2.2.14. Clase de adrese IP.

Exista trei clase de adrese IP (fig.2.2.1.4) şi următoarele adrese rezervate: *adresele de reţele* (cele care au adresa nodului 0, de exemplu 129.0.0.0), *adresele broadcast* (cele care au biţii din adresa nodului setati la 1) şi *adresele loopback* (127.0.0.0 şi 127.0.0.1).

Folosind rutere, reţelele pot fi divizate în subreţele (fie din motive de distanţă, fie din motive de reducere a încărcării din cerinţe de izolare a unei reţele sau de creşterea securităţii); în acest caz, porţiunea de adresă nod a adresei IP este divizată în două părţi: <adresa IP> = <Adresa reţea> <Adresa subreţea> <Adresa nod>. O mască de subreţea indică modul de divizare al adresei de nod (fig.2.2.1.5).

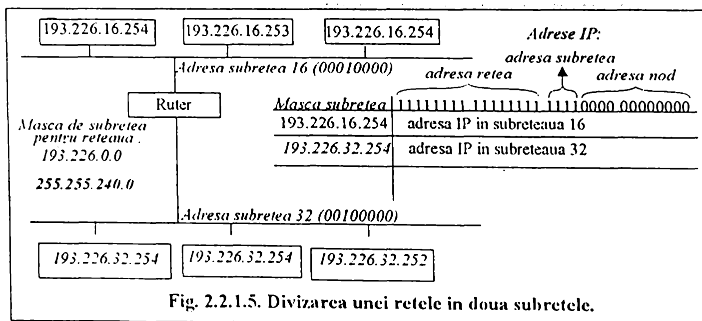


Fig. 2.2.1.5. Divizarea unei retele in doua subretele.

Se prezintă, în continuare, în acest subcapitol, pe scurt, noul protocol Internet (Internet Protocol=IP) asa cum este documentat in RFC 1752 *The Recommendation for the IP Next Generation Protocol.* [Hin96]. Recomandarea a fost aprobată de IESG (Internet Engineering Steering Group) în 17 noiembrie 1994, şi propusă ca standard RFC în decembrie 1995 de către IETF (Internet Engineering Task Force). Denumirea oficială aleasă a fost Internet Protocol version 6 (IPv6, versiunea protocolului IP folosita curent este 4); noua versiune, desemnată ca o revoluţie dar nu ca o radicală schimbare faţă de IPv4, este construită pe arhitectura IPv4 şi are scopul de a rezolva cerinţele legate de extinderea impresionantă actuală a Internet-ului, problemele de rutare, ca şi rezolvarea unor noi cerinţe luate în considerare, precum securitatea, autoconfigurarea, servicii în timp real.

BUPT

1.Cerinte impuse unui nou protocol Internet şi realizate de IPv6

Există o serie de cerinţe pe care trebuie să le respecte un nou protocol Internet, rezolvate în totalitate de IPv6, motiv pentru care acesta a fost desemnat ca standard al noii generaţii de protocoale Internet (de fapt, IPv6 este rezultatul evoluţiei mai multor propuneri de protocoale la IETP, între anii 1992-1994, (*The Internet Protocol*-PIP, *The Simple Internet Protocol*-SIP, TP/*IX*, *TCP and UDP with Bigger Addresses-TUBA*, *IP Address Encapsulation*-IPAE, *Simple Internet Protocol Plus*-SIPP-, *Common Architecture for the Internet-CATNIP*.[Hin96]). Iată citeva dintre aceste cerinţe: (1)*compatibilitate cu IPv4*; IPv6 a fost proiectat astfel încît să poată interactiona cu IPv4. Mecanisme specifice protocolului IPv4 (adrese stil IPv4, reguli pentru checksum) au fost inglobate şi în IPv6 în scopul realizarii tranziţiei de la IPv4 la IPv6; (2)-*facilităţi extinse în ceea ce priveşte posibilităţile de adresare şi de rutare.* IPv6 măreşte dimensiunea adreselor IP, (de la 32 biţi la IPv4) la 128 de biţi, în scopul furnizării mai multor nivele ierarhice de adresare, a unui număr foarte mare de noduri adresabile şi a unei mai simple configurări de adresă. IPv6 introduce un nou tip de adrese, *anycast address* pentru a identifica orice nod dintr-un set de noduri la care poate fi trimis un pachet cu o astfel de adresă de destinaţie, tehnică utilă pentru controlul traseului traficului. Pe de altă parte, IPv6 adaugă un cîmp, *scope*, adreselor multicast îmbogătind astfel gama de adresare şi permite folosirea unui *mecanism de adrese locale* care face posibilă instalarea *plug and play*. Structura adreselor adoptată de IPv6 este realizată astfel încît să poată exprima şi adresele altor protocoale precum IPX şi NSAP, facilitînd migrarea de la aceste protocoale la IPv6.; (3)-*adaugarea de noi funcţionalităţi precum servicii real-time*, autentificare, asigurarea securităţii, integrităţii şi confidentialităţii datelor.

Toate cerinţele de mai sus au fost impuse de amploarea actuală a Internetului (80 000 de reţele în ianuarie 1996, număr care se dublează la aproximativ 12 luni), gama foarte largă de computere pe care le cuprinde (de la PC-uri la supercomputere, majoritatea incluse în reţele locale) ca şi de domenii de interes pe care le acoperă (începind de la institutii de invatamint -de la scoli primare pina la universitati- şi companii comerciale şi terminind cu domenii nou intrate în sfera Internetului sau în curs de a intra precum reţele locale bazate pe frecvenţe radio, pentru care se pun probleme specifice, datorită celor citeva ordine de marime mai mici ale vitezei de transmisie impuse de disponibilitatea frecvenţelor, alocarea spectrului, ratei de erori crescute sau canalele de televiziune, video la cerere, care avind în vedere diminuarea diferentelor dintre computere şi televiziune tind să devină o imensă piata pentru Internet ce va necesita astfel un protocol dotat cu mari posibilităţi de rutare şi adresare ca şi de autoconfigurare [Hin96]).

2.Structura pachetelor Headere IPv6

IPv6 este proiectat ca fiind succesorul lui IPv4; protocolul consta în două părţi: *header-ul de baza*, prezentat în figura 2.2.1.6. şi *headerele extensiilor IPv6*. Multe dintre extensii nu sunt examinate sau procesate de ruter-ele de pe traseul pachetelor; aceasta se realizează numai la sosirea la destinaţie (spre deosebire de IPv4, unde fiecare ruter era obligat să analizeze toate opţiunile). O altă îmbunătăţire faţă de IPv4 este aceea ca dimensiunea header-elor extensiilor IPv6 este variabilă, iar dimensiunea lor totală într-un pachet nu este limitată la 40 de octeţi. Pentru a menţine alinierea la multiplu de 8 octeti, header-ele extensiilor au întotdeauna o dimensiune multiplu de 8 octeti. Header-ele extensiilor definite pînă la ora actuală se referă la următoarele opţiuni IPv6: rutare, fragmentare şi reasamblare, autentificare, securitate date, opţiunea *hop-by-hop*, opţiuni pentru destinaţie (care urmează să fie analizate la nodul destinaţie).

<table>
<tr><td>Versiune</td><td>Prioritate</td><td colspan="2">Eticheta traseu</td></tr>
<tr><td colspan="2">Lungime</td><td>Header-ul urmator</td><td>Limita hop</td></tr>
<tr><td colspan="4">Adresa sursa
(16 octeti)</td></tr>
<tr><td colspan="4">Adresa destinatie
(16 octeti)</td></tr>
</table>

Fig.2.2.1.6. **Formatul unui header IPv6**

Versiune=cimp de 4 biţi în care se inscrie numărul versiunii (6).

Prioritate =cimp de 4 biţi care conţine valoarea priorităţii.

Eticheta traseu-cimp de 24 de biţi; semnificatia să este legata de facilitatea *calităţii serviciului*.

Lungime cimp de 16 biţi care conţine lungimea restului pachetului (care urmează după header) exprimată în octeti, ca un intreg fără semn.

Header-ul următor =cimp de 8 biţi care identifică printr-un selector tipul header-ului care urmează imediat după header-ul IPv6.

Limita hop =cimp de 8 biţi care conţine un intreg fără semn ce se decrementează la fiecare nod care transmite mai departe acest pachet. Pachetul este descarcat dacă limita atinge valoarea 0.

Adresa sursa=cimp de 128 de biţi care conţine adresa nodului care a emis initial pachetul.

Adresa destinaţie =cimp de 128 de biţi care conţine adresa nodului destinaţie al pachetului (dacă este prezent un header pentru optiunea de rutare, acest cimp poate să nu conţina adresa nodului final).

BUPT

3.Adrese IPv6

IPv6 asociaza adresele (exprimate pe 128 biți) interfețelor (sau seturilor de interfețe) și nu nodurilor. O aceeași interfață poate avea asignate mai multe adrese IPv6, de orice tip. Sunt definite următoarele trei tipuri de adrese:

A.adrese unicast: este cazul în care o adresă corespunde unei singure interfete, astfel încît un pachet trimis la o adresa unicast este recepționat numai de interfața corespunzatoare acelei adrese. Exista mai multe formate de adrese unicast asignabile în IPv6. Acestea sunt: adrese unicast *global_provider_based (tip 1), geographic_based (2), NSAP (3), IPX (4), size_local_use (5), link_local_use (6) [i adrese unicast pentru noduri IPv4 (7)*.

Adresele tip 1 se folosesc pentru comunicațiile globale. Formatul lor este prezentat în figura 2.2.1.7.:

010	REGISTRY ID	PROVIDER ID	SUBSCRIBER ID	SUBNET ID	INTERFACE ID
3 biti	n biti	m biti	o biti	p biti	125 - n -m -o -p biti

Fig. 2.2.1.7.. Formatul unei adrese unicast tip 1.

Adresele unicast tip 5 și 6 au semnificație numai pentru rutare locală (la nivel de SUBNET sau SUBSCRIBER). Formatul lor este prezentat în figurile 2.2.1.8. și 2.2.1.9.:

1111111010	0	SUBNET ID	INTERFACE ID
10 biti	n biti	m biti	118-n-m biti

Fig. 2.2.1.8. Formatul unei adrese unicast tip 5.

1111111010	0	INTERFACE ID
10 biti	m biti	118 -n biti

Fig. 2.2.1.9. Formatul unei adrese unicast tip 6.

Pentru ambele tipuri de adrese (5 și 6), INTERFACE ID este un identificator unic în domeniul în care este utilizat; în majoritatea cazurilor se utilizează o adresă pe 48 de biți IEEE-802 MAC; cimpul SUBNET ID identifică o subrețea specifica, astfel încît combinația SUBNET ID + INTERFACE ID formează o adresă ce va fi utilizată local într-o rețea particulara tip Internet, fără a fi nevoie de vreo alocare de adrese anterioară. Aceasta utilizare locală permite organizațiilor neconectate încă în Internet să opereze fără a avea nevoie de un prefix de adresă din spațiul global de adrese Internet; dacă, ulterior, aceste organizații se conectează la rețeaua globală Internet, pot utiliza în continuare cimpurile SUBNET ID + INTERFACE în combinație cu un prefix global, REGISTRY ID + PROVIDE ID + SUBSCRIBER ID, pentru a genera o adresă globală.

Compatibilitatea IPv6 cu IPv4 în ceea ce privește adresele se realizează prin prevederea formatului de tip 7 (adrese IPv6 compatibile cu IPv4) prezentat în figura 2.2.1.10., dar și printr-un alt format de adresă IPv6, folosit numai pentru a reprezenta adresele nodurilor IPv4 (care nu suporta IPv6) ca adrese IPv6; acest tip de adresa este numit "*IPv4-mapped IPv6 address*" *și* reprezentat în figura 2.2.1.11.

0000....0000	0000	Adrese IPv 4
80 biti	16 biti	32 biti

Fig. 2.2.1.10. Adrese IPv 6 compatibile IPv 4.

0000....0000	FFFF	Adresa IP. 4
80 biti	16 biti	32 biti

Fig. 2.2.1.11. Adrese IPv 6 mapate IPv 4.

B.Adrese anycast; este cazul în care o adresă se asignează unui set de interfete (in mod tipic apartinind unor noduri diferite). Un pachet trimis la o adresa anycast va fi recepționat de una dintre interfețele desemnate, și anume de cea mai "apropiată", masurarea distanței fiind făcută de protocoalele de rutare.

Adresele anycast se alocă din spațiul de adrese unicast, utilizînd oricare din formatele descrise mai sus. Astfel adresele anycast sunt sintactic indistingibile de cele unicast; în schimb, atunci cind o adresa unicast se atribuie mai multor noduri (transformîndu-se în adresă anycast), nodurile respective trebuie configurate explicit pentru a "cunoaște" că sunt noduri anycast.

Adresele anycast, atunci cînd sunt utilizate într-o secvență de rutare, permit unui nod să-și aleagă traseul pe care să-l urmeze pachetul (*source selected policies*); în acest caz, o adresa anycast corespunde unui furnizor de servicii Internet (PROVIDER ID).

C.Adrese multicast; este cazul în care o adresă se asignează unui grup de interfete. Orice interfață poate aparține la oricîte grupuri multicast. Formatul unei adrese multicast este descris în figura 2.2.1.12.:

11111111	000T	SCOP	GROUP ID
8 biti	4 biti	4 biti	112 biti

Fig. 2.2.1.12. Formatul unei adrese multicast.

Semnificatia cimpurilor este următoarea: - *T* indică o asignare permanentă (T=0) sau temporara (T=1).- *SCOP* selectează printr-o valoare pe 4 biți gama de vizibilitate a adresei multicast. Sunt recunoscute următoarele codificari: 0 = Rezervat, 1 = Vizibilitate la nivel de nod, 2 = Vizibilitate la nivel de *link*, 3,4 Neasignat, 5 = Vizibilitate la nivel de *site*, 6,7 = Neasignat, 8 -D=vizibilitate la nivel de organizație, E-Vizibilitate la nivel global, F = Rezervat.

În IPv6 nu există adrese broadcast; funcția lor a fost preluată de multicast.

BUPT

B. Protocolul TCP

Protocolul TCP acceptă mesaje de lungime arbitrară pe care le transmite în segmente de maxim 64Ko; fiecare octet transmis are asociat un număr de secvenţă de 32 biţi. Structura unui segment TCP este prezentată în fig.2.2.1.13. Elementele caracteristice ale protocolului sunt prezentate în fig.2.2.1.14 printr-o comparaţie cu protocolul de clasă 4 de la nivelul transport OSI (TP4 OSI).

<table>
<tr><td colspan="8">32 biti</td></tr>
<tr><td colspan="7">Port sursa</td><td>Port destinatie</td></tr>
<tr><td colspan="8">Numar de secventa (pentru fiecare octet)</td></tr>
<tr><td colspan="8">Numar de confirmare</td></tr>
<tr><td>Lungime header TCP</td><td>URG</td><td>ACK</td><td>EOM</td><td>RST</td><td>SYN</td><td>FIN</td><td>Fereastra</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="7">Suma de control</td><td>Pointer de urgenta</td></tr>
<tr><td colspan="8">Optiuni</td></tr>
<tr><td colspan="8">Date</td></tr>
</table>

Fig. 2.2.1.13. Structura unui segment TCP

Semnificatia indicatorilor:

URG= setat la 1 dacă se utilizează Pointer de Urgenta; acesta indică deplasamentul faţă de octetul cu numărul de secvenţă curent la care se gasesc datele urgente (expeditive este termenul OSI) datele urgente sunt cele care trebuie transmise inaintea celor bufferate, de exemplu CTRL/C, CTRL/D, CTRL/S

ACK= setat la 1 indică folosirea cîmpului Numar de confirmare

EOM= indică Sfarsit de mesaj

RST= folosit la resetarea unei conexiuni (de exemplu în cazul recepţionarii unui SYN duplicat sau a căderii unui nod)

SYN= folosit pentru stabilirea conexiunilor

FIN= folosit la incheierea unei conexiuni

Semnificatia porturilor:

Adresele porturilor sursa, destinaţie plus adresele IP corespunzatoare formează o conexiune. Asocierea unui proces la o pereche (port, adresa IP) se realizează cu ajutorul unui *soclu (socket)*. O conexiune cuprinde 2 socluri.

Fig. 2.2.1.14. Elemente caracteristice ale protocolului TCP

Element caracteristic	TCP	OSI TP4	Semnificatie
Numar de tipuri de segmente	1	9	În TCP se foloseşte acelaşi header la toate segmentele . În OSI se folosesc 9 tipuri de headere
Conexiuni permise intre aceleaşi două adrese TSAP	1	2	În TCP o conexiune se identifica printr-o pereche de TSAP, astfel încît este permisă o singură conexiune full duplex; în OSI sunt permise 2 conexiuni full-duplex independente intre aceleasi două TSAP
Format adrese	adresa pe 32 biti	nedefinit	deosebire în formatul adreselor
Calitatea serviciilor	optiuni specifice	negociere	În TCP nu exista negocierea calităţii serviciilor din OSI
Date în cererea de stabilire conexiune	nepermise	permise	În OSI se permite deschiderea conexiunii, de exmplu, în funcţie de valoarea unei parole, precizata în cimpul de date din CR; în TCP nu se permite transfer de date la stabilirea conexiunii
Flux	la nivel de octet	la nivel de mesaj	TCP furnizează un flux de octeti; OSI un flux de mesaje
Date importante	date urgente	date expeditive	TCP foloseste cîmpul Pointer de urgenta, pentru a indica, un numar de octeti, urgent de transmis, din segmentul curent. OSI are doua fluxuri independente de mesaje, multiplexate, date obisnuite şi expeditive.

BUPT

Control explicit al fluxului	Întotdeuna	Uneori	TCP utilizează un control explicit al fluxului permitand receptorului sa insoteasca o confirmare cu o fereastra care specifica domeniul de numere de secventa în care transmitatorul poate furniza date fara a permite o noua confirmare
Confirmari cumulate	Da	Nu	TP4 poate utiliza o schema cu credite
Închiderea conexiunii	Fara pierdere de date	Abrupt	Recuperarea datelor după o închidere abrupta a unei conexiuni se face în nivelul sesiune în OSI; în TCP se foloseste un protocol cu confirmare în 3 paşi pentru închiderea unei conexiuni

În TCP stabilirea şi conexiunilor se face printr-un algoritm cu confirmare în 3 paşi conform diagramei de stari din fig. 2.2.1.15. Stabilirea unei conexiuni presupune două funcţii de deschidere conexiune: una pasiva de aşteptare a unei cereri de stabilire conexiune şi una activa de cerere de stabilire conexiune. Diagrama este asemanatoare cu cea prezentată în sectiunea 2.1.3.

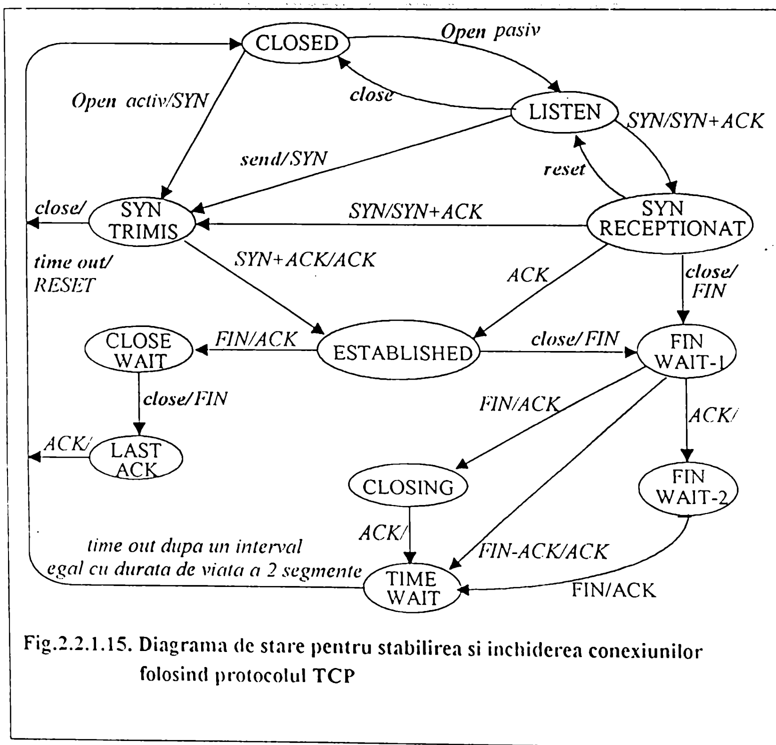


Fig.2.2.1.15. Diagrama de stare pentru stabilirea si inchiderea conexiunilor folosind protocolul TCP

BUPT

C. Comunicarea prin socluri în rețele UNIX și Microsoft Windows (Win32)

Soclurile sunt un mecanism de comunicație interprocese în rețele UNIX, introdus de 4.3 BSD ca o extensie sau generalizare a comunicației prin conducte ("pipes"). Conductele facilitează comunicarea între procese de pe același sistem; soclurile facilitează comunicarea între procese de pe sisteme diferite sau de pe același sistem. O dată cu introducerea comunicării prin socluri și în System V Release 4, soclurile au devenit cea mai importantă interfață de programare (API) pentru rețelele Unix. Deoarece foarte multe companii de calculatoare precum Sun Microsystem, Digital Equipment Corporation și Tektronix au adoptat această interfață de programare soclurile au devenit *de facto* un mecanism standard de comunicație interprocese în rețele Unix. Structura subsistemului de comunicație prin socluri, într-un sistem de operare Unix, este prezentată în figura 2.2.1.16 [Bac89].

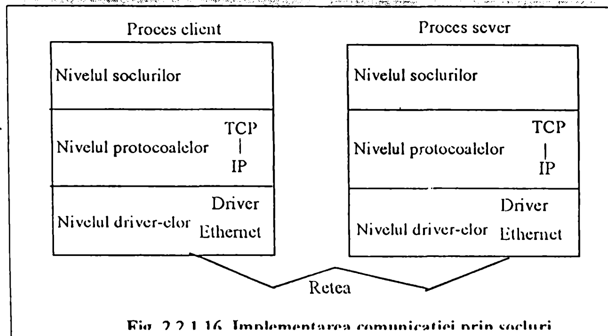


Fig. 2.2.1.16. Implementarea comunicației prin socluri

Subsistemul conține trei părți: nivelul soclurilor, nivelul protocoalelor și nivelul driverelor. Nivelul soclurilor furnizează interfața dintre apelurile sistem și nivelele inferioare. Nivelul protocoalelor conține modulele utilizate pentru comunicație (TCP și IP), iar nivelul driverelor conține driverele care controlează plăcile de conectare în rețea.

Comunicarea între procese prin socluri utilizează modelul client-server (vezi 3.1.1.): un proces server (*listener*) ascultă pe un soclu (un capăt al unei linii de comunicație între două procese) - iar un proces *client* comunică cu serverul, care poate fi situat pe un alt nod, la celălalt capăt al liniei de comunicație, pe un alt soclu . Nucleul sistemului de operare memorează date despre linia de comunicație deschisă între cele două procese [Mus96j].

Se poate face o analogie între apelurile sistem pentru lucrul cu fișiere în UNIX și structurile de date pe care le generează nucleul pentru gestiunea fișierelor cu apelurile sistem pentru comunicația între procese prin socluri și structurile de date pe care le menține nucleul relativ la liniile de comunicație. Astfel, analog descriptorilor de fișiere care se obțin de la sistemul de operare ca urmare a unor apeluri *open()* sau *create()* există descriptori de socluri care se obțin ca urmare a unui apel sistem *socket()*. Nucleul Unix alocă descriptorii de socluri în aceeași tabelă de descriptori în care alocă descriptorii de fișiere[Bac89]. Deci o aplicație nu poate avea un descriptor de fișier și un descriptor de soclu de aceeași valoare. Când o aplicație

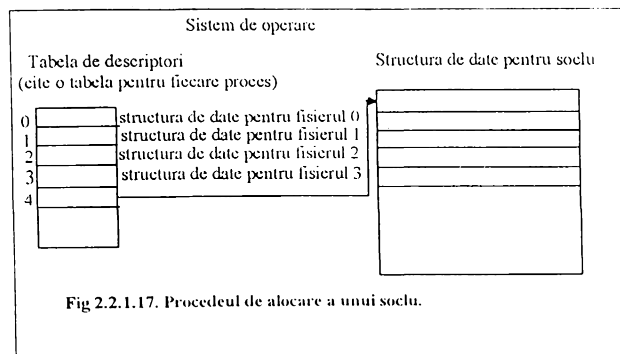


Fig 2.2.1.17. Procedeul de alocare a unui soclu.

BUPT

apelează *socket*(), nucleul alocă o structură de date pentru noua linie de comunicație și completează într-o intrare liberă din tabela de descriptori un pointer la această structură de date. Figura 2.2.1.17 exemplifică acest procedeu. În urma apelului *socket*(), structura de date creată este completată doar parțial, aplicația fiind obligată să apeleze în continuare și alte funcții pentru a completa în întregime această structură de date inainte de a utiliza soclul [Mus96j][Mus91b].

Soclurile care au caracteristici comune în ceea ce privește comunicația, precum convenții de nume, formatul adreselor utilizate de protocoalele de comunicație se grupează în ***domenii*** (familii). Există astfel, pentru comunicația în rețele, ***Internet domain*** și *Xerox NS domain*, iar pentru comunicația între procese de pe același sistem Unix, *Unix system domain*. Fiecare soclu are un tip specificat care determină modul de transfer al datelor, în mod obișnuit, printr-un ***circuit virtual*** (transfer orientat pe conexiune, STREAM SOCKET în terminologia Berkeley), sau prin ***datagrame*** (transfer neorientat pe conexiune). Un circuit virtual permite transferul garantat (bidirecțional) al datelor, fără duplicarea și cu secvențierea pachetelor. Datagramele nu garantează nici transmisia în secvență a pachetelor nici neduplicarea lor, dar au avantajul că nu necesită stabilirea conexiunii. Mecanismul de comunicație prin socluri asociază implicit anumite protocoale de comunicație pentru fiecare combinație ***domeniu -tip soclu*** validă. De exemplu, protocolul TCP furnizează un transfer orientat pe conexiune (circuit virtual), iar UDP furnizează un transfer bazat pe datagrame.

Cuplul de informații ***{domeniu, tip_soclu, protocol}***, relative la un soclu se furnizează sistemului ca parametri ai apelului *socket*() care returnează un descriptor de soclu. Soclul astfel creat nu are însă asociat nici un proces (nu i se asociază la creare nici adresa sistemului local nici a celui de la distanță). Pentru a specifica adresa proprie la un capăt al liniei de comunicație între două procese, procesul server trebuie să execute apelul sistem ***bind()***. Cînd se creează un soclu nu se specifică dacă este activ (pentru a fi utilizat de un client), sau pasiv (pentru a fi utilizat de un server). Serverele care folosesc o transmisie orientată pe conexiune trebuie să execute (după *bind*()) apelul sistem ***listen()*** pentru a pune soclurile lor într-o stare pasivă, gata să primească cereri de stabilire a unei conexiuni. Majoritatea serverelor constau într-o bucla infinită în care preiau o cerere, o tratează și returnează un raspuns. Chiar dacă tratarea cererii durează foarte puțin, cîteva milisecunde, se poate ca o altă cerere să sosească exact în acest interval. Pentru a se evita pierderea cererilor sosite la nivelul serverului, acesta trebuie să transmită apelului *listen*() un argument prin care se cere sistemului de operare să înlănțuie cererile primite pe un soclu. Astfel, argumentele apelului *listen*() specifică ce soclu se setează ca pasiv și dimensiunea maximă a cozii de cereri pentru acel soclu. În scopul recepționarii cererilor de conectare de clienți, un server trebuie să execute în continuare un apel ***accept()***. ***Accept***() creează cîte un soclu nou pentru fiecare cerere de conexiune și returnează apelantului descriptorul noului soclu creat. Serverul utilizează noul soclu exclusiv pentru comunicația cu clientul respectiv, în timp ce pe vechiul soclu acceptă noi cereri de conectare.

Referitor la client, acesta trebuie să execute un apel ***connect()*** pentru a stabili o conexiune cu un server. Clientul trebuie să furnizeze sistemului informații despre identitatea serverului (adresa IP și numărul portului, de exemplu). După stabilirea conexiunii, schimbul de informații dintre client și server se poate face cu apelurile ***read() / write()***, iar în final, distrugerea conexiunii se face cu apelul ***close()***. Figura 2.2.1.18 ilustrează secvența de apeluri sistem executată de un server și un client în cazul comunicării prin TCP.

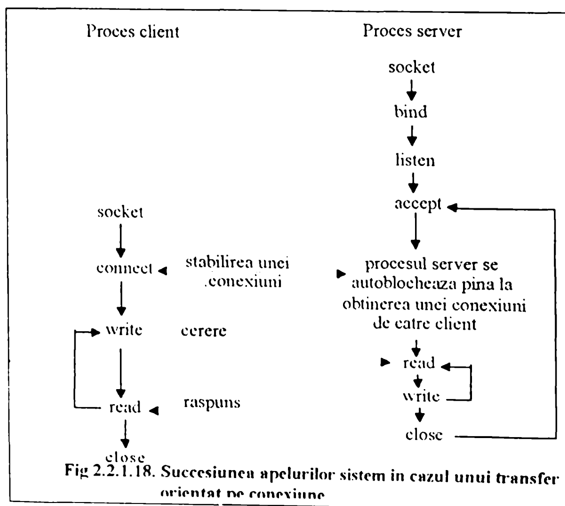


Fig 2.2.1.18. Succesiunea apelurilor sistem in cazul unui transfer orientat pe conexiune

BUPT

Pentru o transmisie prin UDP secvența de apeluri este prezentată în figura 2.2.1.19.

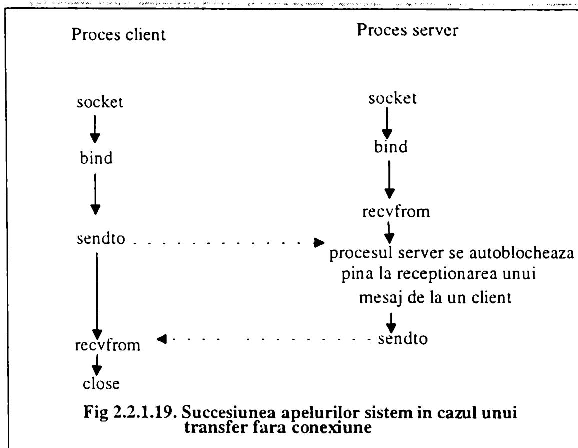


Fig 2.2.1.19. Succesiunea apelurilor sistem in cazul unui transfer fara conexiune

În acest caz, clientul nu trebuie să se conecteze la server; el trebuie doar să trimită către server un mesaj sub forma unei datagrame cu ajutorul apelului sistem ***sendto()*** care primește ca parametru adresa sistemului destinație. Analog, serverul nu trebuie să accepte o conexiune ci doar să recepționeze mesaje datagrame de la un client, cu ajutorul apelului sistem ***recvform()***, care blochează serverul pînă la primirea unui mesaj.

Specificațiile Windows Sockets definesc o interfață de programare pentru rețea in Microsoft Windows bazată pe modelul soclurilor Berkely, standardul de facto TCP/IP. Există însă cîteva API-uri Windows Sockets care nu respectă întru-totul convențiile Berkely, datorită dificultăților de implementare în mediul Windows ; aceste deosebiri se referă la faptul că două funcții de lucru cu soclurile au fost înlocuite complet (*open*() și *ioctl*() cu *closesocket*() și *ioctlsocket*()), funcția *select ()* a fost modificată în sensul că fiecare set de socluri este totuși reprezentat de tipul *fd_set* dar acesta este implementat ca un tablou de tipul SOCKET (tip nou introdus în Windows pentru descriptorii soclurilor), codurile de eroare nu sunt accesibile via variabila *errno* ci via funcția WSAGetLastError(). Pe de altă parte au fost introduse o serie de API-uri noi, care, cu excepția WSAStartup() și WSAClenup() nu sunt obligatorii iar unele sunt disponibile și într-o formă compatibilă cu funcțiile Berkeley, dar sub forma *WSAAsync*XX() ele sunt recomandate în special datorită necesității respectării paradigmelor de programare Windows.

Aplicațiile descrise în anexele B4, B7 utilizează specificațiile Windows Sockets.

BUPT

2.2.2. Familia de protocoale Novell IPX/SPX

Cele două protocoale IPX/SPX au fost introduse de Novell în produsele NetWare. IPX este protocolul de bază Novell NetWare şi este recomandat de Novell ca protocolul de comunicaţie pentru aplicaţiile client-server. Protocolul IPX este o implemetare a protocolului Internetwork Datagram Packet (IDP) proiectat de Xeros Corporation. Este un protocol neorientat pe conexiune şi suportă numai mesaje de tip datagrame. Pe de altă parte, protocolul SPX este orientat pe conexiune; livrarea pachetelor necesită stabilirea unei conexiuni, emisia/recepţia mesajelor printr-un dialog adecvat şi închiderea sesiunii. Astfel, se poate spune că interfaţa de programare în IPX corespunde nivelului reţea iar interfaţa de programare în SPX corespunde nivelului sesiune[Mus96j].

```
unsigned short  checksum;     /*high-low 1's complement*/
unsigned short  packetLen;    /*high-low*/
unsigned char   transportControl;,
unsigned char   packetType;
unsigned long   destNetwork;  /*high-low*/
unsigned char   destNode[6];
unsigned short  destSocket;   /*high-low*/
unsigned long   sourceNetwork; /*high-low*/
unsigned char   sourceNode[6]; /*high-low*/
unsigned short  sourceSocket; /*high-low*/
```

Fig. 2.2.2.1.Structura header-ului pachetelor IPX

Structura unui pachet IPX este indentică cu structura unui pachet Xeros Internetwork Standard (XNS) care constă într-un header de 30 octeţi urmat de 0 pînă la 576 octeţi în zona de date (fig 2.2.2.1.).

Un mesaj poate fi trimis către orice nod din reţea prin plasarea lui în zona de date a unui pachet. Fiecare pachet trebuie să specifice adresa de destinaţie, sub forma unui cuplu {reţea destinaţie, adresă nod, soclu}. Reţeaua destinaţie (cîmpul destNet din structura IPX) este un număr pe 4 octeţi care specifică numărul reţelei locale şi este 0 pentru un nod pe aceeaşi reţea locală ca şi nodul emiţător. Adresa nodului destinaţie (cîmpul destNode din structura IPX) este un număr pe 6 octeţi care conţine adresa fizică a nodului destinaţie şi care poate fi setată la 0x FF FF FF FF FF FF pentru a trimite un pachet broadcast. Soclul destinaţie (cîmpul destSocket din structura IPX) este un cîmp pe 2 octeţi conţinînd adresa soclului procesului destinaţie.

```
unsigned char   connectionControtl;   /*bit flags*/
unsigned char   dataStreamType;
unsigned short  sourceConnectionID;   /*high-low*/
unsigned short  destConnectionID;     /*high-low*/
unsigned short  sequenceNumber;       /*high-low*/
unsigned short  acknowledgeNumber;    /*high-low*/
unsigned short  allocationNumber      /*high-low*/
```

Headerul unui pachet SPX are in plus cele 7 cimpuri de mai sus.

Fig. 2.2.2.2. Structura header-lor pachetelor SPX

Un pachet SPX conţine în plus în header 12 octeţi pentru a urmări secvenţa pachetelor şi numărul confirmărilor (fig 10)şi poate conţine pînă la 534 octeţi de date. Aplicaţiile care transmit pachete SPX trebuie să stabilească conexiuni cu aplicaţiile destinaţie; pachetele transmise pentru care nu s-a primit confirmare sunt retransmise după expirarea unui interval de timeout. După un anumit număr de retransmisii neconfirmate se presupune că aplicaţia destinaţie nu mai "ascultă" sosirea pachetelor şi întrerupe conexiunea.

Înainte de a utiliza fie funcţiile IPX, fie funcţiile SPX, aplicaţiile trebuie să pregătească şi să completeze o structură de date numită EventControlBlock (ECB). Numai după aceea se pot apela funcţiile IPX sau SPX cu pointeri către ECB, care conţine adresa pachetului IPX sau SPX.

Un sistem client-server tipic constă într-un proces server care deschide un soclu (Novell IPX API: *IPXOpenSocket*) şi aşteaptă să recepţioneze un pachet de la orice client (Novell IPX API: *IPXListenForPacket*). Serverul poate de asemenea să "anunţe" prezenţa sa (Novell Bindery API: AdvertiseService) astfel încît atunci cînd un client trebuie să se conecteze la server, el poate să găsească serverul (Novell Bindery API: ScanBindery) şi să determine adresa sa (Novell IPX API: *IPXGetLocalTarget*). În final, clientul poate trimite o cerere de serviciu la server (Novell IPX API: *IPXSendPacket)* [Mus95c][Mus96j][Nov90a][Nov90b].

Aplicaţiile din anexele B3, B, C1, C2, C3,C4 utilizează protocolul de comunicaţie IPX; exemplul din anexa C4 constă într-o aplicaţie client-server într-o reţea Novell.

BUPT

2.3. Aspecte specifice ale suportului de comunicație pentru sistemele de operare distribuite

S-a arătat în secțiunea 2.1.3 că, în ceea ce privește sistemul de operare, importanța cea mai mare o prezintă nivelul transport, care poate oferi funcții apelabile direct din procese. Serviciile și mecanismele necesare pentru a implementa un protocol de transport pot fi clasificate, concluzionînd secțiunea 2.1.3, astfel:

(1) *Mecanisme pentru gestiunea conexiunilor* care implică aspecte capăt-la-capăt în ceea ce privește: (a) alocarea, sincronizarea și dealocarea de resurse; (b) negocierea unor resurse si a unor moduri de desfasurare a operațiilor . Gestiunea unei comunicații *sigure* poate fi obținuta prin următoarele mecanisme: (a) *utilizarea unor algoritmi cu confirmare în 2 sau 3 pași* pentru deschiderea și închiderea conexiunilor, algoritmi apreciati însă drept costisitori ca timp, deoarece necesită schimburi de mesaje (b) *folosirea timerelor*.

(2) *Mecanisme pentru controlul erorilor* care implică aspecte legate de pierderea, duplicarea sau deteriorarea mesajelor; pot fi utilizate următoarele mecanisme: (a) *sume de control* pentru detectarea mesajelor eronate; pachetele eronate sunt ignorate iar recuperarea lor se face prin utilizarea de confirmări și retransmisii; (b) *confirmări explicite* pentru detectarea mesajelor pierdute; o problemă este determinarea time-out-ului; (c) *confirmări implicite* pentru detectarea mesajelor pierdute; mecanismul se bazează pe presupunerea că un raspuns al unui server este confirmat de următoarea cerere sau neconfirmat de time-out; (d) *numerotarea mesajelor și a pachetelor în mesaje*, pentru a detecta mesajele lipsă sau cele care nu sunt în secvență.

(3) *Controlul fluxului* care implică un acord aprioric sau o negociere capăt la capăt asupra mecanismului folosit; pot fi utilizate: (a) ferestre glisante explicite; (b) ferestre glisante implicite (de exemplu cu o singură cerere/raspuns în așteptare la un moment dat); (c) presupuneri privind rata de transmisie și de recepție.

(4) *Stabilirea dimensiunii* mesajelor care diferă de la un protocol la altul și implică segmentarea mesajelor. Se apreciaza ([Gos91][Mul89]) ca pierderile de timp legate de controlul erorilor și fluxului pachetelor în mesaje și de segmentarea/reasamblarea trebuie luate în considerare.

Ca aspecte specifice în vederea implementarii acestor servicii și mecanisme pentru sistemele de operare distribuite pot fi enumerate următoarele:

(1) Primul aspect de care trebuie să se țină seama este ca sistemele distribuite sunt implementate în general în LAN-uri, astfel încît cea mai mare parte a traficului de mesaje rămîne într-o singură rețea locală; chiar dacă sistemul cuprinde mai multe LAN-uri interconectate, proiectanții sistemului încearcă, în general, să facă în așa fel încît traficul dominant să fie cel din rețeaua locală.

(2) Un alt aspect important pornește de la aprecierea dimensiunii datelor care se vor transfera în sistemul distribuit; de exemplu, pentru minimizarea traficului în rețea, bazele de date moderne folosesc tehnici pentru alocarea optima a datelor și alegerea judicioasa a locațiilor care vor rezolva cererile lor. Plecînd de la ideea (firească) că dimensiunea datelor transferate într-un sistem distribuit este legată de dimensiunea fișierelor existente în sistem (in ultima instanță ceea ce se transfera sunt fișiere de date, parțial sau total) și realizîndu-se o statistică a dimensiunii fișierelor și capacității de memorare ocupată de ele în [Mul89] se ajunge la următoarea concluzie: dacă fișierele unui file-server sunt citite cu aceeași frecvența, 95% din timpul de transfer se consuma pentru citirea unor fișiere cu dimensiunea mai mare decît a unui pachet (presupunînd că un pachet are o dimensiune sub 2Ko). Acest lucru se poate constata din fig.2.3.1 unde numai 5% din capacitatea de memorare este ocupată de fișiere sub 2Ko; fișierele mai mari de 24Ko care reprezintă 10% din totalul fișierelor ocupa mai mult de 60% din capacitatea de memorare, iar fișierele mai mari de 1Mo, deși puține (0,1% din totalul fișierelor) ocupa totusi 17% din capacitate.

Concluzia este că proiectarea unui protocol de transport trebuie facuta astfel încît să dea rezultate atît pentru transferuri voluminoase cît și pentru transferuri mici. Ca urmare, timpul necesar pentru pregatirea unui transfer (rezervarea resurselor pentru a obține o productivitate mai buna) trebuie să fie mic pentru a nu contribui la scăderea eficienței transferurilor mici și pe de altă parte eficiența transmisiilor voluminoase trebuie să fie mare deoarece aceste transmisii ocupa o fracțiune mare din timpul total de transfer. Rezultă că sunt necesare două tipuri de protocoale: unul pentru transferuri mici, cu *întîrzieri minime* și altul pentru transferuri voluminoase cu *productivitate maxima*. Este exact ceea ce s-a standardizat: două clase de protocoale, una orientată pe conexiuni (pentru transferuri voluminoase) și alta neorientată pe conexiuni (pentru transferuri mici). Exista însă și

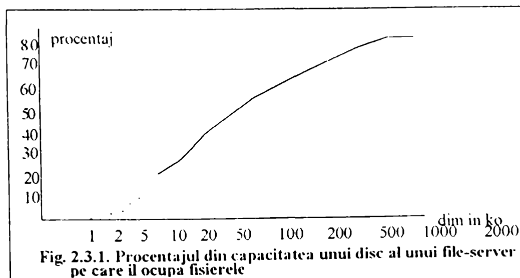


Fig. 2.3.1. Procentajul din capacitatea unui disc al unui file-server pe care il ocupa fisierele

BUPT

sisteme distribuite [Mul89][Gos91] pentru care s-a proiectat un singur protocol de transport eficient atît pentru mesaje mici cît şi pentru mari.

(3) Legat de aspectul (2) este şi următorul: o mare parte din comunicaţii se datorează transmiterii unor cereri de execuţie operaţii la distanţă (de exemplu se cere unui server de la distanţă crearea unui fişier sau a unei ferestre) şi se aşteaptă rezultatul, care ca şi cererea este un mesaj de dimensiune foarte mică; deoarece, pentru continuarea execuţiei, procesul client este blocat în aşteptarea rezultatului, este clar că trebuie minimizată întîrzierea în preluarea rezultatului.

(4) Un alt aspect este cel legat de tratarea erorilor. Erorile de tip date corupte, pierdute sau afară din secvenţă pot fi recuperate prin retransmisii; erorile grave precum căderea unui nod, a unei linii de comunicaţii pot fi recuperate mult mai greu. Primul tip de erori este tratat în OSI la nivelul transport; erorile din al cel de-al doilea tip sunt recuperate la nivelul sesiune. În sistemele distribuite, al caror suport sunt reţelele locale, clasificate de cele mai multe ori în tipul A de reţele (sigure), frecvenţa erorilor din ambele tipuri de mai sus este cam de acelaşi ordin de mărime şi de valoare, relativ mică; de aceea nu se tratează la nivele diferite. La nivelul cel mai inalt, cel de aplicaţie, se va testa intotdeauna execuţia cu succes a unei operaţii la distanţă.

În vederea proiectării unor protocoale de comunicaţii *rapide*, se prezintă următoarele tehnici:

(1) *Integrare protocol-planificator.* S-a reliefat mai sus, la (2) şi (3) importanţa minimizarii timpului de obţinere a rezultatului unei cereri; în [Mul89] se arată că în sistemul de operare Amoeba, într-o staţie SUN conectată intr-o reţea Ethernet, din aproximativ 1.4ms, timpul de obţinere a raspunsului la cerere nula, 400µs reprezintă transmisia mesajelor pe liniile de comunicaţie plus depunerea lor în memorie şi generarea intreruperilor, alte 500µs timpul de execuţie al protocoalelor şi alte 500µs timpul necesar comutarii procesorului la procesele client şi server. Masurătorile au fost facute în situaţia în care protocolul este incapsulat într-un proces de reţea separat; în aceasta situaţie procesorul trebuie comutat la fiecare recepţie şi emisie. O implementare eficientă impune includerea codului protocolului în nucleului sistemului de operare şi parţial chiar în rutinele de intrerupere dacă este posibil, reducînd astfel la minim comutarea contextului.

(2) *Controlul fluxului între un emiţător rapid şi un receptor lent* poate fi făcut prin introducerea unei mici întîrzieri inainte de transmiterea unui pachet; aceasta întîrziere poate fi determinată chiar în timpul transmisiilor. Metoda a fost implementată în protocolul VMTP (Versatile Message Transport Protocol), care se bazează pe asa numitele *pachete în rafala, packet blasts*. În VMTP o datagrama mare este descompusă în grupuri de cel mult 32 pachete; fiecare pachet dintr-un grup este insotit de un număr de secvenţă de 32 biţi, dintre care numai unul are valoarea 1. La recepţie, numerele de secvenţă sunt verificate printr-o operaţie OR; dacă toate pachetele au fost recepţionate, toti biţii rezultatului trebuie să fie 1; dacă sunt biţi 0, pachetele corespunzatoare vor fi retransmise (*retransmisie selectiva*). Presupunem că, avînd un emiţător mai rapid decît receptorul, la transmisia pachetelor dintr-o datagrama, fiecare al 2-lea pachet se pierde. Mecanismul de retransmisie va retransmite printr-o altă rafală (blast), astfel încît ar urma să se piarda pachetele multiplu de 4 s.a.m.d. Pentru a ocoli acest lucru, VMTP lucrează cu o întîrziere între pachete, *interpacket gap*, initial aceasta este setată la 0 de către emiţător dar cînd se constată că s-au pierdut pachetele pare se măreşte gap-ul pină cînd nu se mai pierd pachete (s-a constatat că valoarea intîrzierii este în jurul a cateva sute de microsecunde).

(3) *Utilizarea datagramelor mari şi a unei facilităţi de scater-gather.* Permiţînd datagrame mari, se poate reduce semnificativ overhead-ul de transmisie. În plus operaţiile de copiere ale header-ului şi a datelor unui pachet într-un/dintr-un buffer pot fi eliminate dacă se permite ca acestea să fie în zone diferite de memorie (scater-gather); mai mult, se poate ţine seama şi de structura cererilor: de exemplu, Amoeba permite ca parametrii unei cereri pentru un server de fişier (cod operaţie, nume fişier, pozitia în fişier) să fie plasate într-un buffer separat de bufferul de date. Evitarea operaţiilor de copiere a mesajelor din/in zona de date a protocolului şi a celei a receptorului/emiţătorului poate fi facuta şi cu ajutorul mecanismului de gestiune a memoriei, prin partajarea paginilor. Apar complicaţii legate de faptul că mesajele trebuie aliniate la graniţe de pagini iar o pagină poate conţine numai mesaje complete; aceasta inseamnă că compilatoarele trebuie să rezerve bufferele de mesaje respectînd aceste conditii.

(4) *Utilizarea timer-elor de analiză.* Avînd în vedere următoarele aspecte ale utilizarii timer-elor: (a) startarea şi oprirea unui timer necesită timp. S-a aratat că mai multe timere pot fi implementate folosind un singur timer fizic şi o lista sortată după valorile de time-out, astfel încît startarea şi oprirea unui timer- inseamnă operaţii de inserare şi stergere dintr-o lista, evident cu dezactivarea intreruperii de timer în timpul operaţiei; (b) timerele expiră arareori, numai cînd mesajele sunt pierdute, la avaria unui nod sau a unei linii de comunicaţii; (c) valorile la care sunt setate timerele sunt utilizate pentru recuperarea erorilor; ele nu trebuie să fie extraordinar de exacte - se poate concluziona că timerele pot fi inlocuite cu un fir separat al nucleului, activat periodic, al cărui rol este de a analiza structurile de date ale protocolului (la care trebuie să poată avea acces); el compară starea curentă a protocolului cu cea anterioara şi dacă nu se constată un progres, initiaza procedura de serviciu (de exemplu retransmisia unui pachet). Acest fir se execută cu o prioritate mică în background.

BUPT

2.4. Protocoale cu caracter universal şi protocoale cu caracter restrîns

Protocoalele de transport proces-către-proces continuă să constituie un domeniu important de cercetare în cadrul sistemelor distribuite avînd în vedere faptul că protocoalele de transport stau la baza implementării primitivelor de comunicație interprocese în sistemele de operare distribuite. Exista două direcții în cadrul acestor cercetari: (1) Proiectarea şi implementarea unor protocoale specifice unui sistem de operare distribuit; (2) Reexaminarea proiectării şi implementării protocoalelor cu caracter universal.

Pentru ambele tipuri de protocoale există argumente pro şi contra. Astfel, *protocoalele cu caracter universal* oferă avantaje precum portabilitate, independența aplicațiilor, dar şi dezavantaje care provin din ignorarea aspectelor specifice ce apar în sistemele distribuite, prezentate în secțiunea anterioară; aceste dezavantaje constau într-o complexitate excesivă (protocoalele de transport la nivelul ISO/OSI sunt un exemplu în acest sens) cu repercursiuni asupra performanțelor, orientarea preponderentă pe variante de orientate pe conexiune, lipsa unui suport pentru implementarea unor facilități speciale în rețelele locale precum partajarea resurselor multicasting.

Protocoalele cu caracter restrans au ca principal avantaj performanțe mai bune, (întîrzieri mici, printr-o execuție eficientă) dar şi dezavantajul aplicabilității numai într-un anumit tip de rețele si/sau arhitectură de aplicații. Aceste protocoale cu caracter restrîns se obțin în urma luării în considerare a următoarelor simplificări:

(1) minimizarea modului orientat pe conexiune; în acest sens, există următoarele soluții: (a) utilizarea unui protocol neorientat pe conexiune ori de cîte ori este cazul; (b) utilizarea aşa numitelor protocoale cu conexiuni "lightweight", în cadrul carora o conexiune este deschisă de un server la recepția primei cereri a unui client, şi închisă la recepția unei indicații de sfîrşit de transfer; (c) stabilirea unei conexiuni numai pe timpul necesar satisfacerii unei cereri

(2) minimizarea numărului de pachete schimbate la stabilirea conexiunilor şi cofirmarea lor pe baza unor presupuneri asupra caracteristicilor de eroare ale rețelei

(3) minimizarea circulației confirmărilor prin considerarea recepției raspunsului la o cerere sau a recepției unei cereri imediat după o altă cerere drept confirmări (presupunînd că nu există pachete duplicate sau în afara secventei).

În ceea ce priveşte *optimizarea protocoalelor cu caracter general* se enumeră următoarele soluții:

(1) Utilizarea unei alte arhitecturi pe nivele. În [Gos91], în urma analizei mai multor sisteme de operare distribuite contemporane se propune arhitectura din fig.2.4.1.

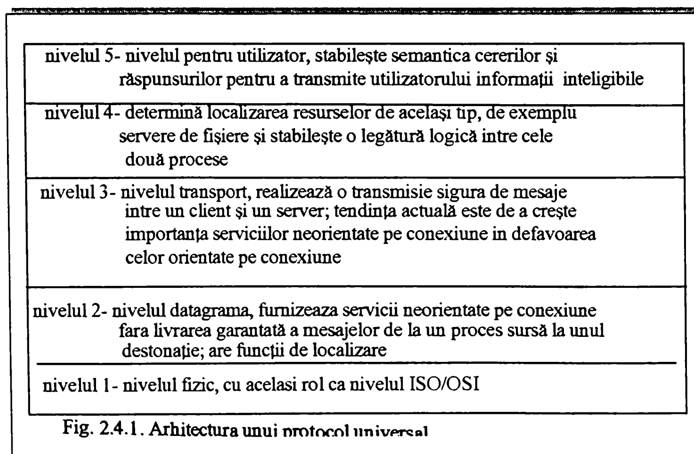


Fig. 2.4.1. Arhitectura unui protocol universal

(2) Implementarea protocoalelor parțial în sistemul de operare, parțial în drivere şi hardware; procesele utilizator pot să interactioneze cu nivelul transport, celelalte 3 nivele mai inferioare sunt însă neaccesibile.

(3) Suport pentru transferul direct al datelor direct din spațiul de memorie utilizator în rețea şi invers.

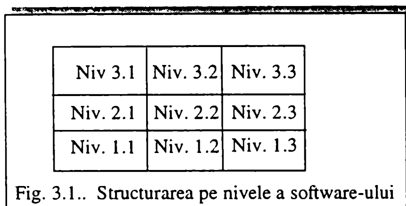
BUPT

Capitolul 3.

PARADIGME ALE INTERACTIUNII DINTRE PROCESE IN SISTEME DISTRIBUITE

În mod uzual, software-ul de sistem şi de aplicaţie au o structură modulara, ierarhizată, pe nivele (fig. 3.1); software-ul de sistem ocupă nivelele inferioare, iar cel de aplicaţie nivelele superioare.

Într-un sistem distribuit, o aplicaţie poate fi structurată în mai multe procese, repartizate pe mai multe procesoare, dispuse pe mai multe noduri într-o reţea. Ca atare, se pot identifica două clase de interfeţe între procese [Mul89]: interfeţe între procese de pe nivele diferite şi interfeţe între procese din acelaşi nivel.



Niv 3.1	Niv. 3.2	Niv. 3.3
Niv. 2.1	Niv. 2.2	Niv. 2.3
Niv. 1.1	Niv. 1.2	Niv. 1.3

Fig. 3.1.. Structurarea pe nivele a software-ului

În sistemele traditionale, nedistribuite, interfeţele între nivele includ interfaţa cu sistemul de operare (incluzînd sistemul de gestiune fişiere, ca un subset), interfeţele cu diferite biblioteci de sistem şi interfeţele cu un sistem de gestiune ferestre. Mai mult chiar, în unele sisteme, interfaţa cu sistemul de gestiune ferestre apare ca o interfaţă între procese care comunică printr-o reţea (de exemplu, sistemul X-window, MIT). Pe de altă parte, fiecare nivel din structura generală din fig. 3.1 se poate spune că oferă servicii nivelului superior. Nivelul defineşte o anumită clasă de obiecte şi operaţii asupra acestor obiecte. Astfel, sistemul de operare furnizează ca obiecte: procese, fişiere, directoare etc; sistemul X-window furnizează obiecte precum ferestre, hărti de biţi etc; o bibliotecă matematică furnizează obiecte precum numere complexe, matrici etc. În mod traditional, interfaţa dintre nivele a fost implementată procedural (interfeţele cu bibliotecile sunt în general bazate pe apeluri de proceduri; interfeţele cu sistemele de operare sunt dotate cu proceduri *stub*) sau sub formă de comunicare prin mesaje (Microsoft Windows). Într-un sistem distribuit, procesele din diverse noduri trebuie să comunice prin intermediul mesajelor transmise prin mediul fizic de comunicaţie. Interfaţa dintre procesele situate în noduri diferite poate avea însă acelaşi format ca şi interfaţa dintre procesele locale. Performanţele subsistemului de comunicaţie între procese depinde, pe de o parte, de primitivele de comunicaţie între procese (interfaţa de comunicaţie) şi pe de altă parte, de protocolul de comunicaţie, care furnizează aceste primitive. Cele mai cunoscute clase de primitive utilizate pentru comunicaţia între procese în sistemele de operare distribuite sunt: (1) *comunicarea prin mesaje,* care se asociază modelului client/server şi la care fluxul de mesaje poate fi unidirecţional sau bidirecţional; (2) *apel de procedură la distanţa (RPC),* care oferă un mecanism puternic tipizat pentru transferul bidirecţional al informaţiei; (3) *tranzactii,* care oferă un mecanism complex pentru executarea şi sincronizarea de operaţii asupra unor obiecte partajate, asigurînd recuperarea informaţiei în cazul aparitiei de erori hardware şi software; (4) *comunicare prin conducte* (*pipes*) ca o generalizare a modelului conductelor locale introdus de sistemele Unix.

Aceste clase de primitive permit implementarea a patru tipuri de procese într-un program distribuit: *filtre, clienti, serveri şi perechi.* Un *filtru* este un proces care transforma datele; el primeşte datele în flux pe canalele sale de intrare, execută prelucrari asupra acestor valori şi trimite apoi rezultatele la canalele sale de ieşire. Exemple binecunoscute sunt o mulţime de comenzi ale sistemului Unix (comenzile care prelucrează fişierele text).

Un *client* este un proces care comută un proces server dintr-o stare pasivă într-o stare activă, transmiţîndu-i cereri. Astfel, un client iniţiază o activitate în momentele alese de el; în majoritatea cazurilor el este întîrziat pînă la indeplinirea serviciului. Un *server* este un proces infinit care, în mod uzual deserveste mai multi clienti.

Procesele *pereche* sunt o colecţie de procese identice care cooperează pentru a furniza un serviciu sau pentru a calcula un rezultat. De exemplu, două procese pereche pot gestiona fiecare o copie a unui fişier şi interacţiona pentru a păstra cele două copii consistente. Un alt exemplu de procese pereche este dat de procesele care interacţionează pentru a rezolva o problemă de calcul paralel, fiecare rezolvînd o parte a acestei probleme.

BUPT

3.1. Modelul client-server

Multe din sistemele de operare distribuite folosesc modelul client/server în scopul repartizării funcţiilor de control în procesele din diverse noduri (fig. 3.1.1).

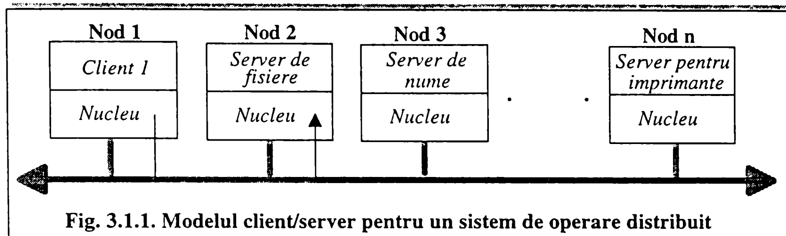


Fig. 3.1.1. Modelul client/server pentru un sistem de operare distribuit

Se poate afirma că există o tendinţă actuală a sistemelor de operare de a plasa cît mai multe funcţii de la nivelul nucleului sistemului de operare la nivelul proceselor server, reducîndu-se astfel nucleul la o formă minimală.

Urmatoarele sisteme de operare distribuite experimentale au fost realizate pe baza modelului client/server[Gos91]: Eden, V Kernel, Amoeba; caracteristica lor principala, comuna, este aceea că resursele sunt controlate centralizat, prin servere.

Se apreciază ([Gos91]) că există trei probleme majore ale modelului client/server: (1) dacă un nod pe care se află unul sau mai multe servere este indisponibil, întregul sistem devine indisponibil; o astfel de soluţie nu oferă un grad prea mare de siguranţă, şi, pentru unele funcţii de control critice într-un sistem (precum cele ale unui server de nume sau de fişiere) poate deveni inadecvată; (2) dacă numărul clienţilor este foarte mare pentru un server, acesta poate intra într-un regim de "sufocare"; cu cît numărul clienţilor creste, cu atît efectul este exacerbat; (3) pentru îmbunătăţirea performantelor modelului client/server se poate folosi, local, un mecanism de caching pentru anumite funcţii ale unui server.

3.1.1. Prezentare generală a modelului client/server

Modelul client-server este cea mai cunoscută şi folosită paradigmă în programarea distribuită. Conform acestei paradigme, un proces *server* (în sensul de servant) îndeplineşte un anumit serviciu pentru un proces *client.* Clientul cere serviciul prin emiterea unui mesaj către server. Serverul, la rîndul lui, execută ciclic urmatoarele acţiuni: recepţionează o cerere de serviciu de la un client, îndeplineşte serviciul şi emite un mesaj de răspuns spre client. Acest model, în care informaţia circulă în ambele sensuri între procesele implicate este, evident, asimetric [Mus96j].

Procesul server este un proces ***persistent***; el există întodeauna în sistem, asteptînd pasiv ca un proces client să-i utilizeze serviciile. Pardigma client-server divizează deci procesele care îşi transferă informaţii în două categorii: client şi server, în funcţie de acţiunea pe care procesul o execută - fie asteaptă o comunicaţie (server), fie iniţiază o comunicaţie (client). De fapt, motivaţia fundamentală a paradigmei client-server provine din cunoscuta problema de "rendez-vous". Modelul client-server rezolvă această problemă impunînd ca dintre oricare două procese care vor să comunice, unul să-şi înceapă execuţia şi să aştepte (un timp nedefinit) pînă la contactarea sa de către alt proces. Soluţia este importantă mai ales în reţelele în care comunicaţia se face prin TCP/IP deoarece TCP/IP nu furnizează nici un mecanism care să creeze automat procese la primirea unui mesaj. Serverul este iniţiat ca un proces autonom în reţea. Structura de bază a unui proces server este

```
server(){
    initializari;
    while (TRUE){
        asteapta_o_cerere(...);
        trateaza_cererea(...);
    }
}
```

Figura 3.1.1. Structura unui proces server (I)

reprezentată în figura 3.1.1.

Această structură de bază sugerează o interfaţa prin datagrame deoarece serverul poate accepta o cerere de la orice client. Este posibil, în această implementare a procesului server, ca deservirea unei cereri să se facă şi în cadrul unei conexiuni şi să necesite un timp destul de mare, deoarece, pe de o parte este necesar să se

BUPT

stabilească un circuit virtual între client şi server şi, pe de altă parte, un proces client poate monopoliza serverul un timp nedefinit. De aceea nodul care implementează serverul ar trebui să memoreze cererile către server care sosesc în timp ce acesta tratează o cerere (într-un context de multiprogramare).

Modelul poate fi însă extins astfel încît să folosească multiprogramarea. Se iniţiază întîi un proces special, ***listener***, a cărui singură sarcină este de a accepta cereri de servicii şi de a le transmite altor procese care deservesc cererile. Structura de bază a procesului server obţinut în această implementare este prezentată în figura 3.1.2.

```
listener(){
   socket client, server;
   virtual_circuit circuit_virtual;
   initializari;
     while (TRUE){
     client  <-- asteapta_o_cerere(socket_impus);
     server <-- creare_socket();
     circuit_virtual <-- accept(client, server);
     creare_proces(server, ..., circuit_virtual, ...);
     }
}
server(...,circuit_virtual, ...){
   while (*serviciul este cerut){
     read(circuit_cirtual,...);
     indeplineste_serviciul(...);
     write(circuit_virtual,...);
   }
   close(circuit_virtual);
}
```
Figura 3.1.2. Structura unui proces server (II)

Fiecare cerere a unui client generează crearea unui nou proces server conectat la procesul client printr-un circuit virtual. Clientul poate monopoliza acest server un timp nedefinit, fără a provoca blocarea altor clienţi care pot emite în acelaşi timp cereri către procesul listener; acesta creeza cîte un proces server pentru fiecare cerere.

În acest al doilea model, procesele server sunt de fapt proceduri identice care se execută asupra unor structuri de date partajate (tabloul de cereri recepţionate). Comutarea contextului între procesele server poate deveni un factor dominant şi perturbator în deservirea cererilor. De aceea modelul poate fi perfecţionat prin folosirea *firelor* ("threads"). Firele nu au alocate resursele necesare unui proces; ele pot opera însă asupra resurselor alocate procesului care îl încapsulează.

```
client()
{
      message msg;
      socket client, server;
      virtual_circuit circuit_virtual;
      ...
      send_datagram(NAME_SERVER, "nume server");
      msg = receive(NAME_SERVER);
      server = msg.adr_server;
      client = creare_socket();
      circuit_virtual = open(client, server);
      while ()
      {
        write(circuit_virtual, ...);
        read(circuit_virtual, ...);
        analizeaza_raspusul();
      }
      close(circuit_virtual);
      ...
}
```
Figura 3.1.3. Structura unui proces client

În figura 3.1.2 clientul îşi transmite cererea la un "*socket impus*", în sensul de o adresă cunoscută în reţea. Astfel, se impune proceselor client să "cunoască" o adresă fixată în reţea, pentru a-şi putea adresa cererile de servicii la acea adresă. Dacă adresa este cunoscută atunci ea se poate specifică direct în programul sursă al clientului- dar, în acest caz , o dată compilat programul client, serverul nu mai poate fi mutat pe alt nod- sau poate fi returnată de un server de nume.

Dacă procesul client intenţionează să trimită o cerere de servicii unui server de nume pentru a proiecta numele serverului dorit într-o adresă atunci serverul de nume trebuie să memoreze un catalog de servere şi adresele lor. Acest scenariu pentru luarea contactului cu un server şi pentru adresarea cererilor este descris în figura 3.1.3.

În diverse aplicaţii software, serverele trebuie să poată avea acces la fişiere, porturi, la care în mod normal sistemul de operare interzice accesul unui client obisnuit. În aceste cazuri serverele se execută cu privilegii speciale şi trebuie luate masuri de protecţie ca aceste privilegii speciale să nu fie transmise clienţilor care apelează aceste servere. De exemplu, un server care se execută ca un program privilegiat trebuie să conţină o funcţie care să verifice dacă un fişier poate fi accesat de un client sau nu.

În general, serverul trebuie să asigure verificari pentru: (1). ***Autentificarea clienţilor*** - verificarea identităţii unui client. (2). ***Autorizarea clienţilor*** - determină dacă unui client îi este permis accesul la serviciul pe care serverul il furnizează. (3). ***Securitatea datelor*** - garantarea păstrării integrităţii datelor (ele pot fi compromise de clienţi în mod intenţionat sau neintenţionat) (4). ***Păstrarea secretului datelor*** - garantarea interzicerii accesului unor clienţi la datele particulare ale unui client. (5).***Protecţia datelor*** - garantarea că aplicaţiile nu pot abuza de resursele sistemului

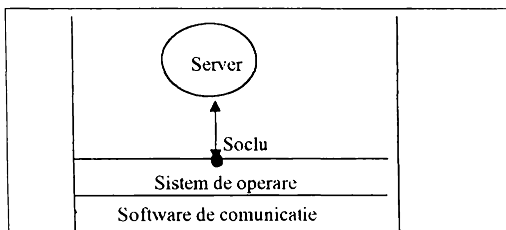
BUPT

3.1.2. Concurenta la nivelul serverelor si clienţilor. Exemple.

S-au prezentat în secţiunea precedentă două modele ale procesului server. Termenul de *server iterativ* desemnează implementarea unui server care procesează la un moment dat o singură cerere, iar termenul de *server concurent* implementarea unui server care procesează mai multe cereri la un moment dat. Se va arăta în acest subcapitol că în cazul multor servere este suficientă realizarea unei concurenţe aparente; astfel, dacă un server execută puţine calcule în raport cu operaţiile de I/0 pe care trebuie să le execute, acesta este posibil să se implementeze ca un singur proces care utilizează operaţii de I/0 asincrone, pentru a permite comunicaţii simultane pe mai multe canale. Din perspectiva unui client, serverul apare comunicînd concurent cu mai mulţi clienţi. În acest sens, termenul de *server concurent* se referă numai la faptul ca serverul gestionează concurent cererile nu şi la implementarea serverului care utilizează procese concurente.

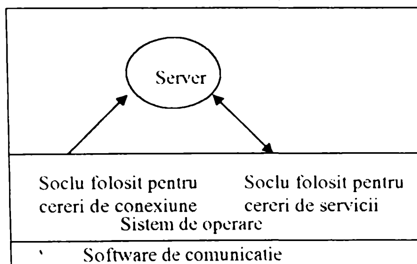
Se poate defini ca *timpul de procesare* a unei cereri de către un server, timpul necesar unui server pentru a trata o cerere izolată, iar ca *timpul de raspuns* întîrzierea produsă de la lansarea unei cereri de către client pînă la primirea răspunsului.

Serverele iterative (fig 3.1.2.1.) procesează cîte o cerere, pe rînd. Dacă o nouă cerere apare în timp ce serverul este ocupat cu tratarea alteia, sistemul memorează nouă cerere; la terminarea tratării cererii curente, se extrage nouă cerere de tratat s.a.m.d.. Dacă lungimea medie a cozii de cereri este N, atunci timpul de răspuns pentru o cerere poate fi apreciat în medie ca fiind mai mare de N/2 +1 ori decît timpul de procesare a unei cereri. Deoarece timpul de răspuns creste proporţional cu N, multe implementări restricţionează valoarea lui N la o valoare mai mică (de exemplu 5) si sugerează ca programatorii să folosească servere concurente în cazul în care o coadă de cereri, de o astfel de dimensiune mică nu este suficientă. Un server iterativ trebuie analizat si din alt punct de vedere: acela al suprasaturării sale. Astfel, un server iterativ desemnat a deservi C clienţi, fiecare trimiţînd K cereri pe secundă va trebui sa aibă un timp de procesare pentru o cerere mai mica decît 1/(C*K) secunde. Dacă serverul nu poate trata cererile cu această rată cerută, atunci există pericolul depaşirii cozii sale de cereri şi deci a pierderii unor cereri. Pe de alta parte, un server iterativ este uşor de proiectat, depanat şi verificat; varianta iterativă poate fi aleasă cînd execuţia iterativă furnizează un răspuns suficient de rapid pentru încărcarea estimată. În mod obişnuit, serverele iterative se asociază unui protocol neorientat pe conexiune, deşi pot fi scrise variante de servere iterative şi bazate pe protocoale orientate pe conexiune.



(a) Server iterativ, protocol neorientat pe conexiune

Algoritm:
(1) Se creaza un soclu si se leaga (bind) acest soclu la adresa stabilita pentru serviciul oferit
(2) In mod iterativ, se citeste cererea urmatoare (recvfrom), se trateaza cererea si se trimite raspunsul clientului



(b) Server iterativ, protocol orientat pe conexiune

Algoritm:
(1) Se creaza un soclu si se leaga (bind) acest soclu la adresa stabilita pentru serviciul oferit
(2) Se asteapta o noua cerere de conexiune pe soclul creat si se obtine un alt soclu
(3) In mod iterativ, se citeste o cerere, se trateaza cererea si se trimite raspunsul clientului
(4) Cand se termina de tratat cererile clientului, se inchide conexiunea (soclul obtinut la doi) si se trece la pasul 2

Fig. 3.1.2.1. Servere iterative
(a) Server iterativ, protocol neorientat pe conexiune
(b) Server iterativ, protocol orientata pe conexiune

Motivul esenţial pentru a utiliza celalalt tip de server, *serverele concurente*, este acela al timpului de răspuns; serverele concurente pot deservi mai multi clienţi oferind un timp de răspuns rapid. Concurenţa îmbunătăţeşte timpul de răspuns dacă: (1) generarea unui răspuns implică operaţii de I/O; prin suprapunerea operaţiilor de I/O corespunzatoare unei cereri, cu execuţia de către procesor a altei cereri, se poate cîstiga timp; (2) timpul de procesare al cererilor variaza mult, in raport de conţinutul cererilor; tehnica *timeslicing* permite tratarea cererilor scurte în acelaşi timp cu cele lungi fara a fi necesară aşteptarea terminării cererilor lungi. (3) serverul se execută pe un calculator cu mai multe procesoare.

Exista două soluţii, conceptual diferite pentru a realiza tratarea concurenta a cererilor intr-un server. O primă soluţie, cea mai cunoscută este de a utiliza mai multe procese sau fire în cadrul serverului. În cadrul unui server neorientat pe conexiuni va exista deci un singur *proces server master* care începe iniţial execuţia; procesul *master* deschide un soclu pe

BUPT

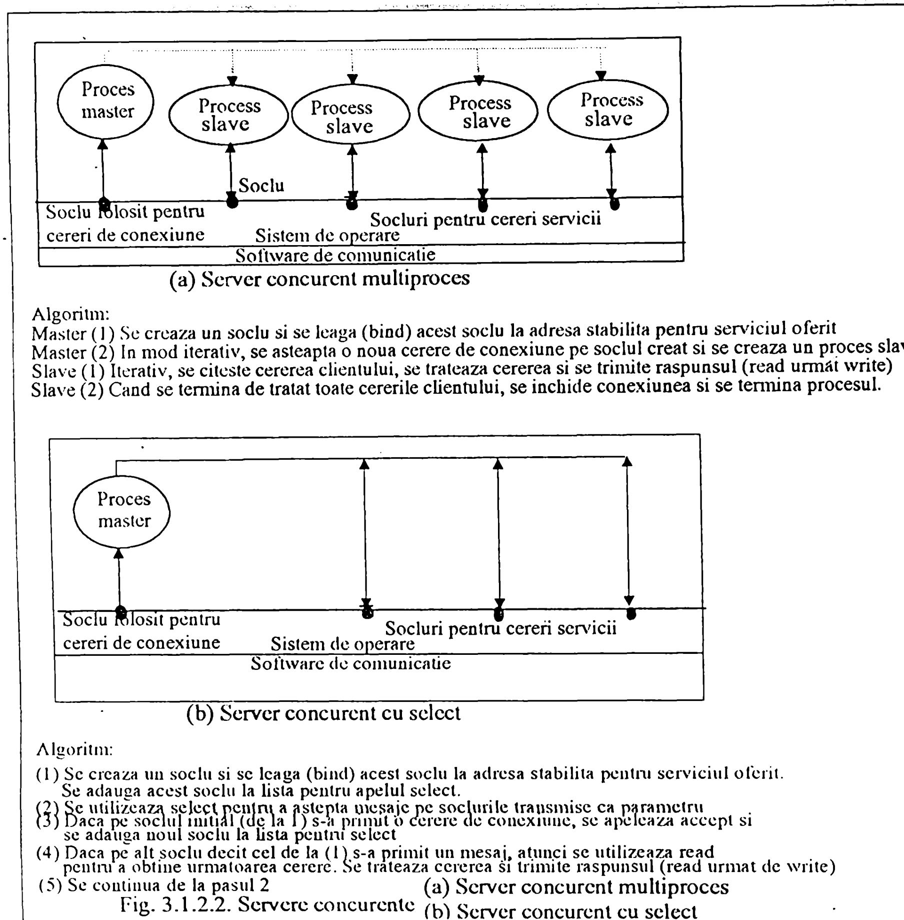


Fig. 3.1.2.2. Servere concurente (a) Server concurent multiproces (b) Server concurent cu select

un port cunoscut în sistem, aşteaptă cerere şi crează un proces *slave* pentru a trata fiecare cerere. Procesul master nu comunică direct niciodată cu un client; aceasta va fi responsabilitatea proceselor slave. Fiecare proces slave comunică cu un client; el îşi încetează execuţia cînd termină de tratat cererea. Deoarece crearea de noi procese este o operaţie costisitoare în timp, serverele neorientate pe conexiune folosesc mai rar implementări concurente. In schimb, serverele orientate pe conexiuni, în cazul cărora o conexiune dintre server şi un client cuprinde mai multe cereri, implementează concurenţa la nivelul conexiunilor şi nu la nivelul fiecarei cereri. Ca şi în cazul unui server neorientat pe conexiune, există un proces master şi procese slave. Procesul master nu comunică direct cu un client niciodată; la orice cerere de deschidere a unei noi conexiuni, procesul master crează un proces slave care să gestioneze conexiunea. In timp ce procesele slave vor interacţiona cu clienţii, procesul master va aştepta alte conexiuni (figura 3.1.2.2.). Pentru multe servicii, este suficient un singur program, care să conţină codul corespunzator atît procesului master cît şi proceselor slave. In cazurile în care scrierea proceselor slave reclamă un program independent, pentru o mai buna proiectare, depanare şi urmarire, în Unix trebuie utilizat un apel *execve* () dupa fiecare cerere de proces (*fork*()).

A doua soluţie pentru a realiza tratarea aparent concurentă a unor cereri dar într-un singur proces, se poate realiza dacă încărcarea totală a serverului (numărul de cereri) nu depăşeşte capacitatea sa de prelucrare. În acest caz, serverul se constituie într-un singur proces care utilizează apelul *select* pentru a gestiona asincron canalele de comunicaţii. Fie, de exemplu, un singur proces server care are deschise mai multe conexiuni TCP pentru mai mulţi clienţi. La sosirea unui mesaj pe o conexiune, procesul se deblochează, tratează cererea şi trimite un răspuns, dupa care se autoblochează, asteptînd din nou sosirea de mesaje pe una dintre conexiuni. Atît timp cît procesul este suficient de rapid pentru a satisface cererile prezentate serverului, această versiune de server este cel puţin atît de rapidă cît şi versiunea serverului multiproces. Ideea de bază este ca într-un singur proces să se înglobeze funcţiile atît ale procesului master cît şi a proceselor slave prin folosirea

BUPT

apelului *select*. Serverul crează cîte un soclu pentru fiecare conexiune pe care trebuie să o gestioneze şi apoi execută *select*() pentru a aştepta sosirea de mesaje pe unul din socluri. Apelul *select*() primeşte ca parametrii setul de descriptori ai soclurilor pe care se aşteaptă mesaje şi transmite o mască de biţi care specifică care descriptori din set sunt gata. Pentru a distinge între operaţii master şi slave, serverul utilizează descriptorii; dacă devine gata un descriptor corespunzător unui soclu master atunci serverul execută aceeaşi operaţie pe care ar realiza-o un proces master: apelează *accept*() pentru acel soclu pentru a obţine o nouă conexiune. Dacă devine gata un descriptor corespunzător unui soclu slave, atunci serverul efectuează aceeaşi operaţie pe care ar realiza-o un proces slave: aplează *read* pentru a obţine mesajul, tratează cererea şi trimite răspunsul (3.1.2.2.b.).

Alegerea între un server iterativ şi unul concurent este dificilă deoarece cererile utilizatorilor, vitezele de prelucrare, posibilităţile de comunicare se pot modifica rapid. Legat de concurenţă, mai trebuie abordat un aspect important: acela al *nivelului de concurenta*, definit ca numărul total de procese pe care le-a generat la un moment dat. Nivelul de concurenţă al unui server variază pe parcursul existenţei lui, dar ceea ce interesează este nivelul de concurenţă al unui server maxim atins. În mod obisnuit, un server concurent, bazat pe un protocol orientat pe conexiune crează cîte un proces pentru fiecare conexiune cerută de un client. Evident, un server nu poate crea însă un numar arbitrar de conexiuni, deoarece, pe de o parte, fiecare implementare TCP limitează numarul de conexiuni active posibile şi, pe de altă parte, fiecare sistem de operare limitează numarul de procese care pot fi create în sistem. Cînd serverul atinge una dintre aceste limite, sistemul va refuza serverului crearea de noi procese. Prin urmare, serverele care crează cîte un proces nou la fiecare cerere, fie de conexiune, fie de serviciu, (care urmează astfel o strategie *demand-driven* cînd nivelul de concurenţă creşte la *cerere*), dacă nu limitează explicit nivelul de concurenţă, prezintă riscuri în funcţionarea într-un sistem de operare cu o încărcare mai mare. Strategia *demand-driven* în ceea ce priveşte concurenţa cererilor, desi prezintă avantajul economisirii resurselor (bufferele, blocurile de control pentru procese nu se aloca decît atunci cînd sunt necesare) poate să nu producă însă întotdeauna rezultate optimale. Astfel, dacă timpul de procesare al unei cereri, *ptime*, este mai mic decît timpul necesar pentru crearea unui proces, *ctime*, versiunea iterativă a unui server poate fi mai performantă dacît versiunea concurentă în ceea ce priveşte întîrzierea medie de deservire a cererilor (figura 3.1.2.3.). Intîrzierea medie care se obtţne pentru procesarea a două cereri de catre serverul iterativ este *3*ptime/2*, mai mica decît *3*ctime/2 + ptime/2*, întîrzierea pentru procesarea a două cereri de catre serverul concurent. Implementarea concurentă a serverului va putea deservi o medie de *1/ctime* cereri pe unitatea de timp, în timp ce implementarea iterativa va putea deservi *1/ptime* cereri pe unitatea de timp; evident că dacă rata de sosire a cererilor *r* este *1/ctime<r<1/ptime* numai o implementare iterativă a serverului functionează corect; cea concurentă pierde mult timp pentru crearea proceselor şi rejectează cereri cînd cozile folosite de protocoalele software ajung la capacitatea maxima. Concluzia este că, în practică, nu trebuie utilizate variante concurente decît atunci cînd timpul de procesare a unei cereri depaşeşte timpul de creare a unui proces.

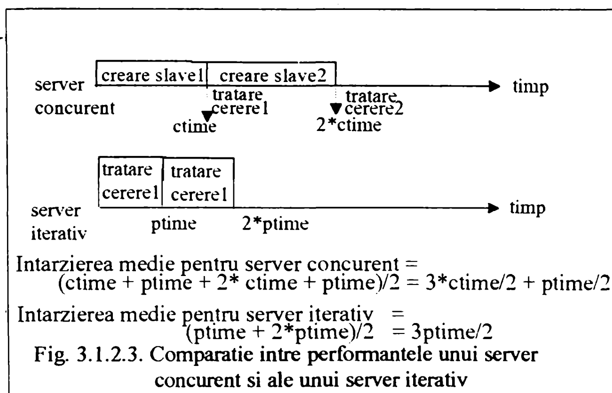


Fig. 3.1.2.3. Comparatie intre performantele unui server concurent si ale unui server iterativ

Pe de altă parte, trebuie cautate soluţii alternative la strategia *demand-driven* în ceea ce priveste concurenţa. O solutie care limitează nivelul maxim de concurenţă al serverelor, controlează întîrzierile în deservirea cererilor şi oferă performanţe chiar şi cînd timpul de creare al proceselor este relativ mare, constă în prelocarea proceselor concurente, pentru a evita pierderile de timp la crearea lor. Astfel, la iniţializare, serverul master crează toate cele P procese slave care îşi încep executia aşteptînd sosirea unei cereri. În Unix, un server master neorientat pe conexiune va crea şi soclul asociat portului rezervat serviciului respectiv; acest soclu va fi moştenit de procesele slave care execută *recvfrom*() pentru a obţine o datagramă şi adresa clientului. Dupa crearea soclului şi a proceselor slave, procesul master se poate termina sau se poate transforma el insuşi în ultimul proces slave, evitînd astfel crearea încă unui proces. Analog se procedează şi cu server master orientat pe conexiune: se crează soclul pe portul rezervat şi se crează apoi cele P procese slave. Acestea executa *accept*() pe soclul moştenit; o cerere de conexiune va debloca un singur slave care va prelua noul soclu, şi va trata cererile pe aceasta conexiune. Cînd clientul termină cererile, procesul slave închide conexiunea şi reia apelul *accept*() pentru a prelua o nouă cerere de conexiune.

Desi prealocarea proceselor poate îmbunătăţi eficienţa, ea nu rezolvă însă toate problemele. Crearea proceselor noi creşte overhead-ul datorat software-ului de comunicaţie. O a doua soluţie, care poate creşte

BUPT

eficiența serverelor este opusă celei precedente, ea constînd în amînarea alocării proceselor. Ideea de bază este următoarea: în loc de a decide a priori o implementare iterativă sau concurentă, se permite serverului sa masoare timpul de procesare a unei cereri și sa aleagă dinamic între o tratare iterativă sau concurentă.

Pentru a implementa dinamic alocarea întîrziată a proceselor slave, algoritmul este urmatorul : serverul master recepționează o cerere, setează un timer (*alarm()* în Unix) și începe tratarea cererii, în mod iterativ. Dacă serverul termină tratarea cererii înainte de expirarea timer-ului, atunci anulează timer-ul ; dacă, însă, timer-ul expira înainte ca serverul să termine tratarea cererii, serverul crează un proces slave care tratează în continuare cererea (apelul *fork*() în Unix permite, pe lîngă moștenirea soclurilor și continuarea codului parintelui exact din punctul în care a fost întrerupt cînd timer-ul a expirat).

Cele două soluții prezentate mai sus pot fi combinate. Un server își poate începe execuția fără a avea procese prealocate și poate utiliza alocarea întîrziată a proceselor. Se așteaptă recepția unei cereri, și se crează un proces slave numai dacă tratarea necesita un timp mai mare decît cel setat la un timer. Dupa ce procesul slave a fost creat și s-a terminat prelucrarea unei cereri, procesul slave nu se termină imediat ci asteapta recepționarea unei noi cereri. Problema soluției combinate nu constă în decizia cînd se crează procese slave ci cînd un proces slave trebuie sau nu terminat. O variantă este ca numarul de procese slave de la un moment dat să nu depășească o limită fixată ; acest lucru poate fi realizat dacă procesele slave partajaza o variabilă (într-o zona de memorie partajată) care indică nivelul de concurență curent și care se testează după tratarea unei cereri pentru a se determina dacă procesul se termină sau persistă. O alta variantă pentru a controla concurența se bazează pe masurarea perioadei de inactivitate a unui server. Algoritmul este urmatorul: procesul slave startează un timer inainte de a trece la asteptarea unei noi cereri; dacă timerul expira inainte de recepționarea unei cereri, procesul slave se termina.

Ca o concluzie a celor prezentate pina acum se pot face următoarele observatii:

1. *Cînd se folosește o varianta iterativă și cand o variantă concurentă a unui server?* Serverele iterative sunt mai ușor de proiectat, implementat și depanat, în schimb serverele concurente furnizează răspunsuri mai rapide la cereri. Se utilizează o variantă iterativă dacă timpul de procesare a unei cereri este scurt și soluția interativă oferă timp de răspuns suficient de rapid aplicatiei respective.

2. *Cand se foloseste un protocol orientat pe conexiune și cand un protocol neorientat pe conexiune?*

Se utilizează un protocol neorientat pe conexiune dacă protocolul de la nivelul aplicației tratează pierderile de mesaje, sosirea pachetelor în secventa (în general, aplicatiile nu realizează, însă, acest lucru) sau dacă procesele client sunt localizate în aceasi rețea locala cu serverul, rețea care nu ridică probleme de fiabilitate.

Dacă clienții și serverul se afla intr-o retea metropolitană se va utiliza intotdeauna un protocol orientat pe conexiune.

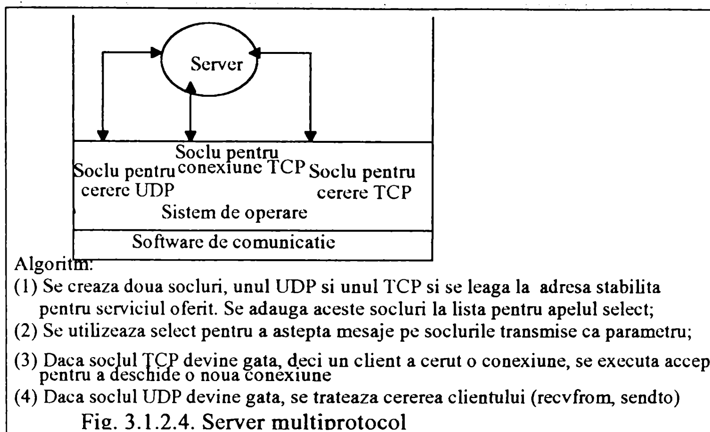
3. *Cînd se folosește o concurență aparentă în server și cînd o concurență reala, multiproces?*

Un singur proces server poate gestiona mai multe conexiuni utilizînd operații de I/O asincrone; un server multiproces folosește sistemul de operare pentru a obține automat tratarea concurentă a cererilor. Un server implementat ca un singur proces (concurență aparentă) se utilizează dacă serverul trebuie sa partajeze sau transmită date între conexiuni. Un server multiproces se utilizează dacă fiecare proces slave poate lucra izolat, obtinîndu-se astfel un nivel mare de concurență (în special pe sisteme multiprocesor).

Folosirea operațiilor de I/O asincrone pentru a gestiona simultan mai multe conexiuni este o soluție și pentru realizarea de servere multiprotocol sau/și multiserviciu. Serverele multiprotocol sunt adecvate pentru situațiile în care un serviciu trebuie realizat atît pentru protocol TCP cît și pentru UDP; dacă același serviciu s-ar implementa în două servere separate (unul UDP, altul TCP) se irosesc resurse ale sistemului în mod inutil (intrări în tabela de procese), mai ales dacă se are în vedere că standardul TCP/IP definește zeci de servicii. Un server multiprotocol constă într-un singur proces care utilizează operații de I/O pentru a gestiona comunicatii, fie pe UDP, fie pe TCP (fig. 3.1.2.4.) și care pentru tratarea cererilor prevede o procedură unica, indiferent dacă cererea a sosit via UDP sau via TCP. Evident că serverul din fig. 3.1.2.4. poate fi extins pentru a permite concurența aparentă între cereri care se receptionează pe mai multe conexiuni TCP sau pe mai multe socluri UDP.

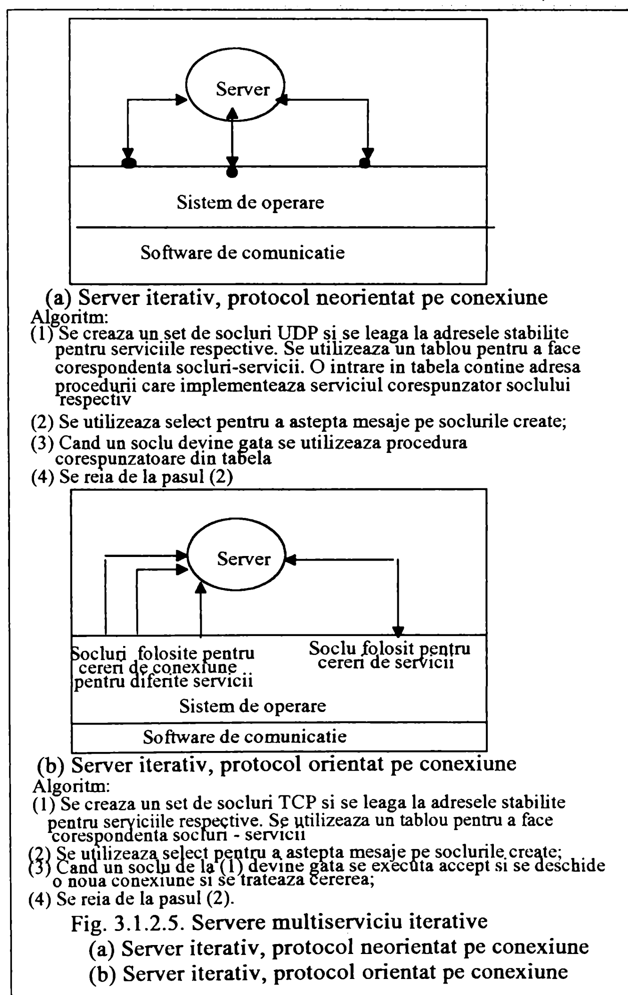
Serverele multiserviciu sunt adecvate pentru situațiile în care inglobarea mai multor servicii intr-un program reduce codul total obținut față de situatia în care aceste servicii ar fi inglobate în servere separate. Serverele multiserviciu pot fi și ele iterative (fig. 3.1.2.5.), concurente (3.1.2.6.) sau chiar multiprotocol sau unele servicii pot fi tratate iterativ și alte concurent, în functie de natura serviciului. Un server multiserviciu multiprotocol este denumit *superserver*.

BUPT



Algoritm:
(1) Se creaza doua socluri, unul UDP si unul TCP si se leaga la adresa stabilita pentru serviciul oferit. Se adauga aceste socluri la lista pentru apelul select;
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile transmise ca parametru;
(3) Daca soclul TCP devine gata, deci un client a cerut o conexiune, se executa accep pentru a deschide o noua conexiune
(4) Daca soclul UDP devine gata, se trateaza cererea clientului (recvfrom, sendto)

Fig. 3.1.2.4. Server multiprotocol



(a) Server iterativ, protocol neorientat pe conexiune

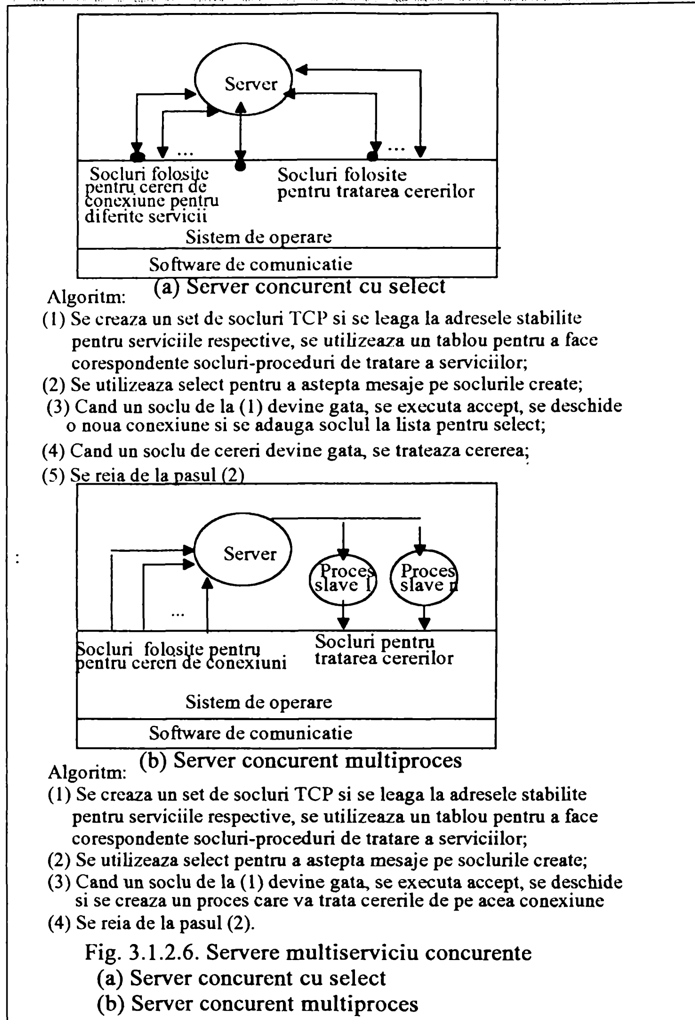
Algoritm:
(1) Se creaza un set de socluri UDP si se leaga la adresele stabilite pentru serviciile respective. Se utilizeaza un tablou pentru a face corespondenta socluri-servicii. O intrare in tabela contine adresa procedurii care implementeaza serviciul corespunzator soclului respectiv
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile create;
(3) Cand un soclu devine gata se utilizeaza procedura corespunzatoare din tabela
(4) Se reia de la pasul (2)

(b) Server iterativ, protocol orientat pe conexiune

Algoritm:
(1) Se creaza un set de socluri TCP si se leaga la adresele stabilite pentru serviciile respective. Se utilizeaza un tablou pentru a face corespondenta socluri - servicii
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile create;
(3) Cand un soclu de la (1) devine gata se executa accept si se deschide o noua conexiune si se trateaza cererea;
(4) Se reia de la pasul (2).

Fig. 3.1.2.5. Servere multiserviciu iterative
(a) Server iterativ, protocol neorientat pe conexiune
(b) Server iterativ, protocol orientat pe conexiune

BUPT



(a) Server concurent cu select

Algoritm:

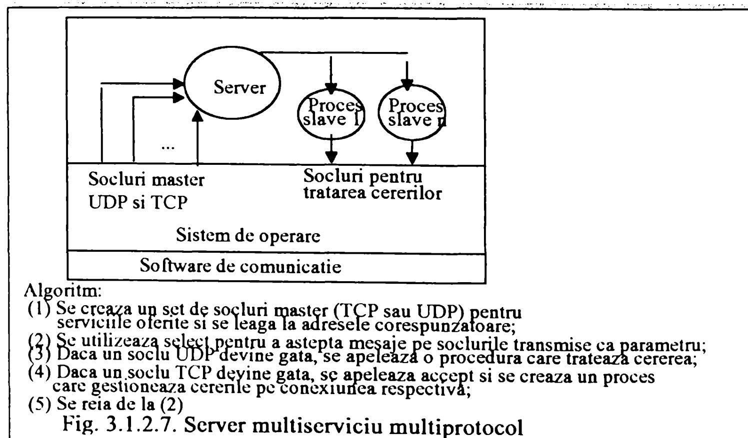
(1) Se creaza un set de socluri TCP si se leaga la adresele stabilite pentru serviciile respective, se utilizeaza un tablou pentru a face corespondente socluri-proceduri de tratare a serviciilor;
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile create;
(3) Cand un soclu de la (1) devine gata, se executa accept, se deschide o noua conexiune si se adauga soclul la lista pentru select;
(4) Cand un soclu de cereri devine gata, se trateaza cererea;
(5) Se reia de la pasul (2)

(b) Server concurent multiproces

Algoritm:

(1) Se creaza un set de socluri TCP si se leaga la adresele stabilite pentru serviciile respective, se utilizeaza un tablou pentru a face corespondente socluri-proceduri de tratare a serviciilor;
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile create;
(3) Cand un soclu de la (1) devine gata, se executa accept, se deschide si se creaza un proces care va trata cererile de pe acea conexiune
(4) Se reia de la pasul (2).

Fig. 3.1.2.6. Servere multiserviciu concurente
(a) Server concurent cu select
(b) Server concurent multiproces



Algoritm:

(1) Se creaza un set de socluri master (TCP sau UDP) pentru serviciile oferite si se leaga la adresele corespunzatoare;
(2) Se utilizeaza select pentru a astepta mesaje pe soclurile transmise ca parametru;
(3) Daca un soclu UDP devine gata, se apeleaza o procedura care trateaza cererea;
(4) Daca un soclu TCP devine gata, se apeleaza accept si se creaza un proces care gestioneaza cererile pe conexiunea respectiva;
(5) Se reia de la (2)

Fig. 3.1.2.7. Server multiserviciu multiprotocol

BUPT

3.1.3. Protocoale pentru modelul client-server

Modelul client-server poate fi implementat utilizînd ambele clase de protocoale prezentate în subcapitolul 2.4: protocoale cu caracter universal şi protocoale cu caracter restrans. O caracteristică a primei clase de protocoale este aceea că prezintă reţeaua de comunicaţie ca pe un dispozitiv de I/O; în consecintă, comunicaţia la nivelul proceselor în reţele se face utilizînd *modelul fişier*. Aceasta înseamnă deschiderea unei conexiuni (open), utilizarea de primitive de citire sau scriere (read, write) şi, în final, inchiderea conexiunii (close). Modelul fişier este propus atît în protocoalele TCP/IP, cît şi în ISO/OSI. S-au aratat în 2.3 şi 2.4 dezavantajele utilizării unui protocol cu caracter general. În sensul obţinerii unei performante superioare, pentru modelul client/server, se poate adopta un protocol cu caracter restrans, a cărui interfaţă cu procesele utilizator consta numai într-un mesaj REQUEST transmis de la procesul client la server, urmat de obţinerea raspunsului de la server, printr-un mesaj REPLY; acest model de comunicaţie se mai numeste şi *modelul operaţiei la distanţa (remote operation model)*. Structura pe nivele a acestui model este foarte simpla (fig. 3.1.3.1).

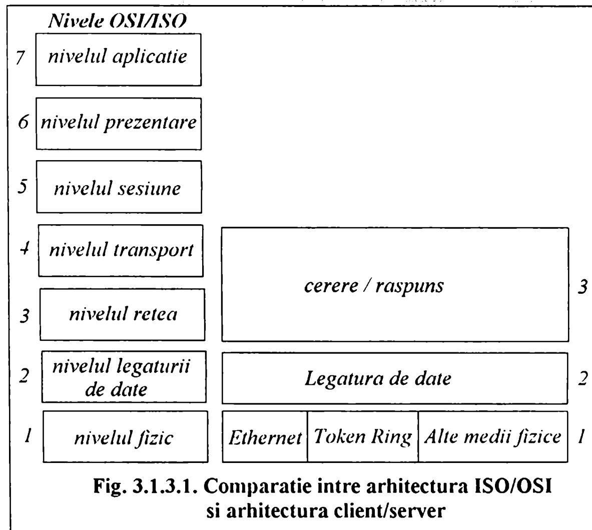


Fig. 3.1.3.1. Comparatie intre arhitectura ISO/OSI si arhitectura client/server

Pentru detectarea şi recuperarea erorilor (folosind, evident, mecanismul confirmarilor) pot fi folosite unele simplificari prezentate în sectiunea 2.3. Astfel, varianta completa de confirmare a ambelor mesaje REQUEST, REPLY (fig. 3.1.3.2) poate fi simplificată în modul urmator: dacă REPLY este primit de client într-un interval de timp fixat, el este considerat şi confirmare; dacă acest interval este depăşit, se retransmite cererea; (3) La primirea unui REQUEST, serverul startează un timer; dacă serverul trimite REPLY fără a depăşi timpul fixat de timer, se consideră acest REPLY ca şi confirmare.

Dacă timerul generează timeout inainte de trimiterea REPLAY-ului, atunci se va trimite o confirmare pentru REQUEST şi apoi REPLY. Se vor utiliza astfel confirmări numai pentru mesajele de REPLY (fig.3.1.3.3); acest lucru este util pentru ca serverele sa execute numai o singură dată o cerere repetata. Dacă nu se confirma de catre client un REPLY şi se retransmite cererea, serverul nu va mai executa cererea ci, constatînd că a primit o cerere deja tratată, va trimite imediat REPLY .

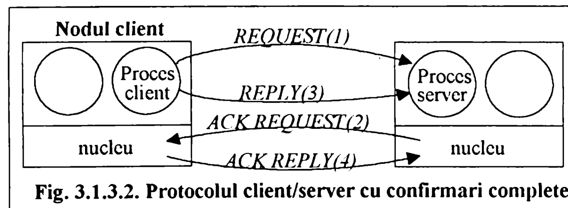


Fig. 3.1.3.2. Protocolul client/server cu confirmari complete

Un alt aspect legat de micsorarea numărului de confirmări este confirmarea la nivel de mesaje, şi nu de pachete (considerînd un mesaj segmentat în pachete) şi transmiterea mesajelor în flux de pachete (mecanismul *packet blasts)*. Un astfel de mecanism este utilizat in protocolul VMTP propus de V kernel.

BUPT

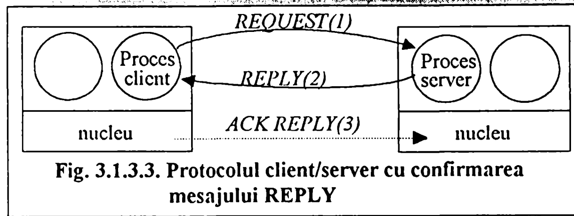


Fig. 3.1.3.3. Protocolul client/server cu confirmarea mesajului REPLY

În [Tan93][Tan96] se apreciază că, pentru controlul erorilor la minimum, sunt necesare şapte tipuri d mesaje (fig. 3.3.4). La nivelul proceselor sunt vizibile, evident, numai REQUEST şi REPLY.

Tip mesaj	De la	Catre	Semnificatie
REQUEST (REQ)	Client	Server	Clientul cere executia unei cereri serverului
REPLY (REP)	Server	Client	Raspuns pentru client de la server
Acknowledg. (ACK)	Oricare	Celalalt	Confirmarea unui mesaj anterior
Are you alive(AYA)	Client	Server	Test pt. un server care nu raspunde la comenzi
Am am alive (IAA)	Server	Client	Raspuns la un test de prezenta
Try again (TA)	Server	Client	Serverul nu poate prelua o cerere (fie nu are loc sa o memoreze, fie nu o poate prelua nefiind intr-o stare in care poate prelua cerer
Address unknown(AU)	Server	Client	Nu exista un proces server asociat acestei adre

Fig. 3.1.3.4. Tipuri de mesaje

În fig. 3.1.3.5 se arată şi cateva protocoale client/server care utilizează aceste mesaje.

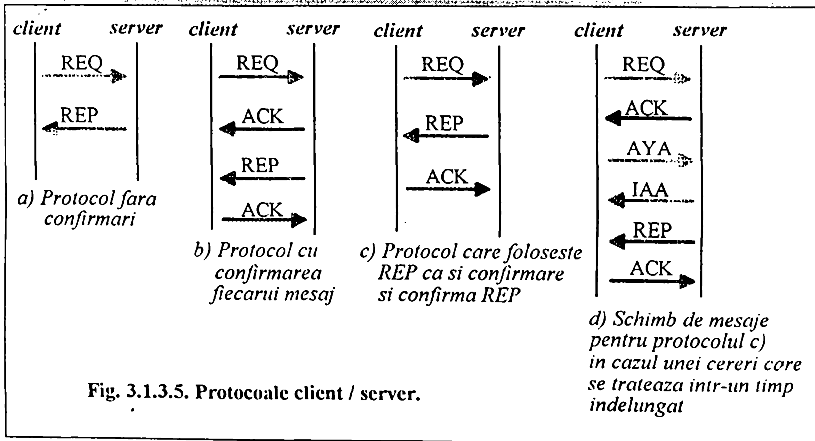


Fig. 3.1.3.5. Protocoale client / server.

BUPT

3.1.4. Sisteme client-server

Se poate considera că, în aplicaţiile distribuite, unul sau mai multi clienţi, şi unul sau mai multe servere, împreună cu suportul pentru comunicarea interprocese formează un sistem compus (Client Server System sau, pe scurt, CSS) care permite efectuarea de calcule şi prelucrări distribuite, analiza şi vizualizarea datelor. Intr-un astfel de sistem, un client este un proces care interacţionează cu utilizatorul şi are urmatoarele caracteristici [Mus96j]:

1. Prezintă o interfaţă cu utilizatorul (User Interface = UI), singurul mijloc prin care utilizatorul îşi poate prezenta cererile de consultare şi regăsire a datelor, dar şi de vizualizare a rezultatelor cererilor sale. În mod tipic, clienţii interacţionează cu utilizatorii printr-o interfaţă grafică (Graphical User Interface = GUI). Deoarece într-un CSS pot coexista mai multi clienţi, pot exista astfel şi mai multe variante de GUI. Unele CSS-uri pot fi prevazute şi cu o interfaţă pentru controlul resurselor sistemului.

Tipic, interfaţa grafică a sistemului de operare defineşte interfaţa GUI dintre client şi utilizator . Astfel, într-un mediu UNIX, sistemul XWindows a devenit, "de facto", interfaţa grafică standard.

2. Formează, pentru una sau mai multe comenzi ale utilizatorului una sau mai multe cereri către server. Serverul şi clientul pot utiliza un limbaj comun pentru prezentarea cererilor. Nu toate comenzile utilizatorului se regasesc, într-o corespondenţă de unu la unu cu cererile adresate serverului.

Un client poate utiliza un mecanism de "caching" şi diverse tehnici de optimizare pentru a reduce fluxul de cereri către server; de asemenea poate realiza verificările legate de autentificare şi autorizare (controlul accesului la date).

3. Comunică cu serverul printr-un mecanism de comunicaţie între procese. Un client ideal ascunde complet utilizatorului mecanismul de comunicaţie interprocese.

4. Execută o analiza a datelor trimise ca raspuns de către server în urma unei cereri şi apoi le prezintă utilizatorului. Complexitatea acestei analize variază de la un CSS la altul.

Intr-un CSS, un server este un proces sau un set de procese, toate aflate pe acelaşi nod care furnizează un anumit serviciu unuia sau mai multor clienţi. Serverul are urmatoarele caracteristici:

a) Un server furnizează un serviciu unui client. Natura şi complexitatea acestui serviciu este definit de scopul fiecărui CSS. Astfel, un server poate efectua prleucrari minore ale datelor (un server de fişier sau un server de imprimante), sau prelucrări complexe ale datelor (un server pentru baze de date sau pentru procesarea imaginilor).

b) Un server raspunde la cererile şi comenzile pe care le primeşte de la clienţii sai. Astfel, un server nu va iniţia niciodată un dialog cu unul dintre clienţii sai. El acţionează astfel ca un depozit de date (server de fişiere), de cunoştinţe (server de baze de date) sau furnizor de servicii (server de imprimante).

Cu toate acestea, un server poate adresa la rîndul său cereri de servicii altui server, deci devine, la rîndul său, un client pentru un alt server. De exemplu, un server de fişier poate cere obţinerea datei şi timpului curent (pe care le asociază unui fişier) unui server de dată şi timp. Situaţia serverului care devine astfel client

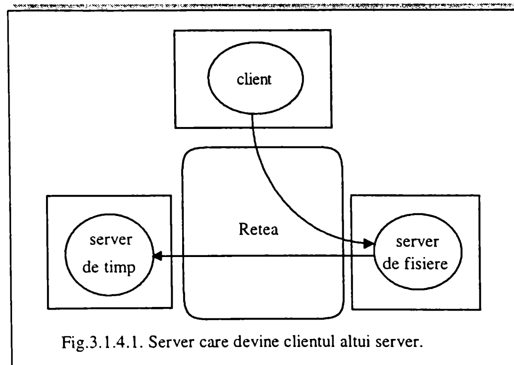


Fig.3.1.4.1. Server care devine clientul altui server.

este ilustrată în figura 3.1.4.1

c) Un server ideal ascunde întreaga componenţă a unui sistem CSS clienţilor şi utilizatorilor sai. Un astfel de server poate comunica cu clienţii săi indiferent de tehnologia de comunicaţie (hardware şi software). Astfel, un client DOS ar trebui să poată comunica cu un server UNIX, indiferent deci de sistemul de operare şi de tehnologia de conectare a clienţilor la server.

BUPT

d) Un server poate memora informaţii relative la interacţiunile sale cu clienţii (*stateful server*) sau poate să nu păstreze nici o astfel de informaţie de stare (*stateless server*). Servere *stateful* sunt motivate de următoarele observaţii:

- prin păstrarea unor informaţii în server se reduce dimensiunea mesajelor pe care clientul şi serverul le interschimbă şi serverul răspunde mai rapid cererilor;
- informaţiile de stare memorate permit serverelor să-şi "amintească" cererea precedentă unei cereri curente şi să construiască un răspuns incremental.

În schimb, serverele *stateless* oferă o fiabilitate mai mare decît serverele *stateful*, pentru că, în cazul ultimelor, informaţiile de stare devin incorecte cînd se pierd sau se duplică mesaje sau cînd nodul clientului este oprit şi repornit.

Ca o concluzie a celor de mai sus se poate spune că arhitectura client-server divide o aplicaţie distribuită în procese separate, localizate pe noduri diferite, conectate într-o reţea formînd astfel ceea ce se ce se cheamă "loosely coupled system". Un proiectant al unei aplicaţii divide aplicaţia sa în subtaskuri care urmează să fie executate fie de procese client, fie de procese server în condiţiile unor restricţii impuse de sistemul CSS însusi şi cu o funcţionalitate dependentă de sistemul de operare. Cu cît API-urile puse la dispoziţia sistemului de operare sunt mai avansate, cu atît codul aplicaţiei va fi mai mic. Pentru proiectantul care dezvoltă o aplicaţie într-o reţea de calculatoare, folosind modelul client-server, prezintă deci o mare importanţă interfaţa de programare (API-urile) pusă la dispoziţie de sistemul de operare. În figura 3.1.4.2. se prezintă plasarea în nivelele modelul OSI a unor astfel de interfeţe de programare cunoscute .

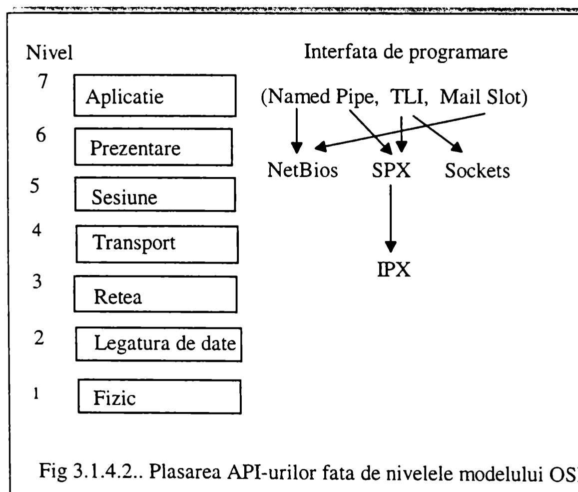


Fig 3.1.4.2.. Plasarea API-urilor fata de nivelele modelului OS

BUPT

3.2. Comunicarea prin mesaje

Comunicarea prin mesaje este asociată cu modelul client/server şi constituie o extensie a comunicării locale interprocese prin mesaje; este o formă de comunicare în care procesele utilizator generează explicit mesajele transmise prin subsistemul de comunicaţie şi utilizează explicit mecanismele de livrare şi recepţie ale mesajelor.

3.2.1. Probleme specifice ale comunicării prin mesaje

Un *mesaj* poate fi definit ca o structura de date compusă dintr-un header de lungime fixa şi o zonă de date de lungime variabila (sau fixă, uneori), care poate fi gestionată de procese, inclusiv livrată catre o destinaţie sau receptionată de la o sursa (fig. 3.2.1.1).

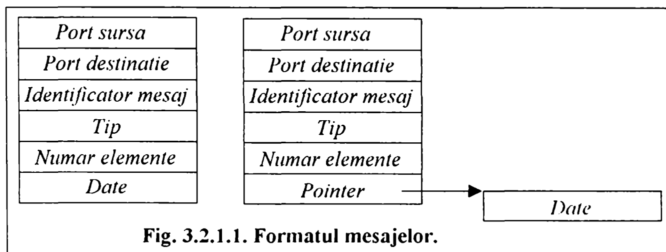


Fig. 3.2.1.1. Formatul mesajelor.

Se enumeră următoarele probleme specifice ale comunicării prin mesaje;
(1) mesajele pot fi complet *nestructurate* sau *structurate*; mesajele structurate sunt de preferat, pentru că: (a) uşurează tratarea unor părţi din mesaj (port sursa, port destinatie) de catre nucleul sistemului de operare şi (b) în sistemele eterogene, numai datelor carora li se ataşează tipul pot fi transmise într-o manieră transparentă;
(2) există două aspecte importante legate de primitivele care implementează comunicarea prin mesaje: stabilirea setului de primitive şi semantica acestora. În fig 3.2.1.2 se prezintă primitivele de bază în comunicarea prin mesaje. Se observă că: (a) clientul specifică prin primitive adresa destinatiei (serverul) şi mesajul şi (b) serverul specifică de la cine primeşte mesajul şi bufferul în care se memorează mesajul receptionat; (c) nu este necesara stabilirea unei conexiuni.
(3) sunt permise în unele implementari şi primitive selective (fig. 3.2.1.2).
(4) semantica primitivelor separă primitivele în primitive *cu şi fără blocare, sigure şi nesigure, cu şi fără bufferare.*

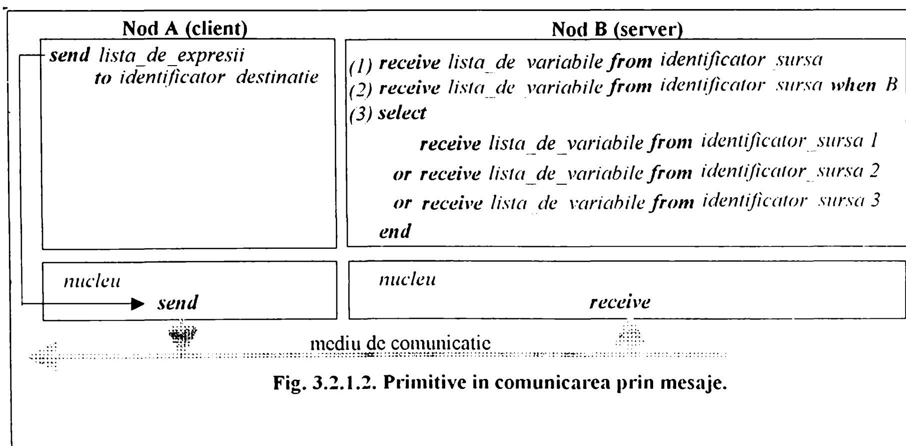


Fig. 3.2.1.2. Primitive in comunicarea prin mesaje.

BUPT

3.2.2. Primitive cu şi fără blocare

O primitiva de comunicare interprocese este *fără blocare (asincrona)* dacă execuţia sa nu determină intîrzierea apelantului; în caz contrar, primitiva este *cu blocare*. Scopul introducerii primitivelor fără blocare este exploatarea paralelismului obţinut datorită acestor primitive (procesul îşii continuă execuţia în paralel cu transmiterea mesajelor).

Semantica *cu blocare, fără blocare* pentru operaţia send se realizează observînd că programul apelant poate obţine controlul dupa execuţia operaţiei *send* în urmatoarele 3 moduri (fig. 3.2.2.1):

(1) semantica *no-wait-send* sau *asyncron-send*- procesul emitator asteapta doar preluarea mesajului de catre subsistemul de comunicaţie;

(2) semantica *syncronization send* - procesul emitator asteapta pînă cînd nodul destinaţie primeşte mesajul;

(3) semantica *rendez-vous* - procesul emitator asteapta pînă cînd procesul receptor primeşte mesajul;

(4) semantica *remote-invocation send* sau *remote procedure call* - procesul aşteaptă pînă cînd primeşte un raspuns de la procesul receptor, în urma tratării cererii.

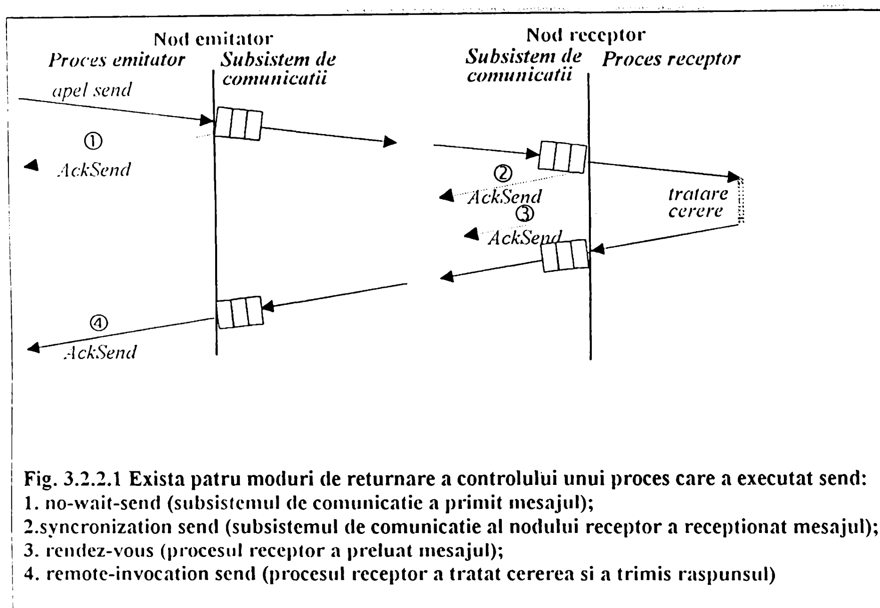


Fig. 3.2.2.1 Exista patru moduri de returnare a controlului unui proces care a executat send:
1. no-wait-send (subsistemul de comunicatie a primit mesajul);
2.syncronization send (subsistemul de comunicatie al nodului receptor a receptionat mesajul);
3. rendez-vous (procesul receptor a preluat mesajul);
4. remote-invocation send (procesul receptor a tratat cererea si a trimis raspunsul)

Modul (2) este mai puţin folosit; pentru o reţea sigură, el este echivalent cu (1). Modul (3) este folosit in sistemele centralizate (dacă procesul receptor a primit mesajul, atunci nu există motive pentru care să nu îl poată prelucra - decît în cazul caderii nodului, dar atunci va dispare şi emitatorul -), dar în sistemele distribuite nu se poate garanta că, dupa primirea mesajului, receptorul va trata cererea şi va returna raspunsul. În practică, se folosesc cele două extreme (1) şi (4) şi uneori (3). Modurile (2)-(4) se consideră cu blocare, iar (1) fără blocare. Unii autori ([Tan96]) clasifică chiar şi modul (1) în categoria primitivelor cu blocare, considerînd fără blocare o primitiva *send* care primeşte controlul imediat; în acest caz, nu pot fi utilizate două primitive *send* consecutive, fără ca apelantul să se asigure că subsistemul de comunicaţie are bufferul liber pentru a prelua cererea. De aceea, pentru acest mod (complet fără aşteptare la execuţia unui *send*) subsistemul de comunicaţie anunţa apelantul printr-o întrerupere că bufferul său este disponibil pentru un nou *send*.

În ceea ce priveste operaţiile de *recepţie fără blocare* se obişnuieşte să se furnizeze aplicaţiilor urmatoarele variante:

(1) o primitiva *receive* care livrează subsistemului de comunicaţie adresa la care se depune mesajul receptionat; pentru a se determina dacă s-a recepţionat sau nu un mesaj se foloseşte o altă primitiva sau se genereazǎ o întrerupere la primirea mesajului;

(2) o primitiva *receive* care preia un mesaj, dacă există disponibil, şi intoarce un cod de eroare dacă nu există un mesaj (eventual, dupa un mic timp de aşteptare); aceasta se completează fie cu o primitiva *wait*, care permite aşteptarea sosirii unui mesaj, fie cu o primitiva *test*, care permite testarea stării operaţiei de receptie;

(3) o primitiva care livrează subsistemului de comunicaţie adresa unei proceduri ce se va executa în momentul sosirii unui mesaj.

BUPT

Fig. 3.2.2.2 arată un transfer de mesaje folosind primitive cu blocare.

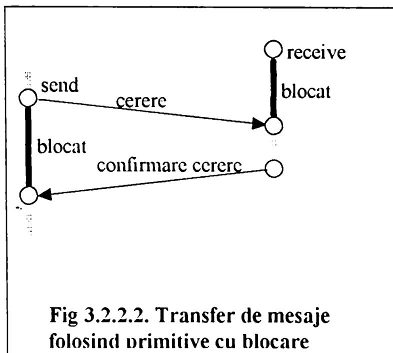


Fig 3.2.2.2. Transfer de mesaje folosind primitive cu blocare

Ambele clase de primitive, cu şi fără blocare, au avantaje şi dezavantaje. Comparativ cu primitivele cu blocare, cele fără blocare oferă posibilitatea paralelismului în execuţie, o maxima flexibilitate şi pot fi folosite şi în aplicaţiile în timp real, dar au şi dezavantajele că conduc la programe nestructurate, greu de urmarit, dependente de timp şi necesita bufferare atît la nivelul receptorului, cît şi la nivelul emitatorului. Unele sisteme oferă pentru emisie varianta fără blocare, iar pentru recepţie varianta cu blocare. În sistemele care oferă variantele cu blocare, paralelismul poate fi obţinut prin folosirea firelor: fiecare cerere receptionată se tratează printr-un fir separat. Firele pot fi create dinamic, la primirea cererilor sau se poate menţine un ansamblu de fire, create static, la lansarea în execuţie (dar în acest caz, pot fi tratate simultan un număr fixat de cereri - egal cu numărul firelor); structurile de date comune firelor trebuie accesate în regim de excludere reciprocă (deci trebuie protejate prin semafoare sau alte mecanisme).

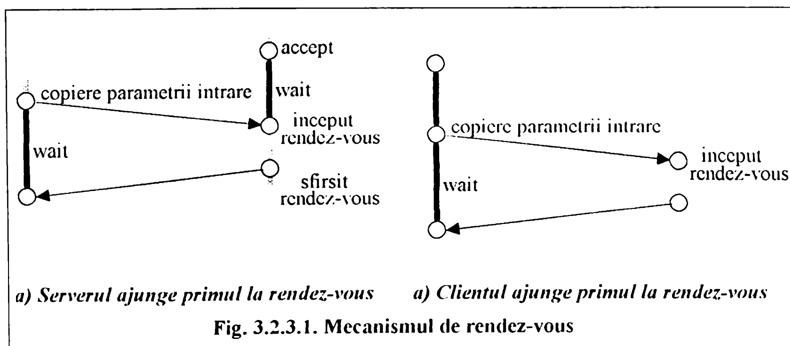
Programele din anexa A3 (client şi server) exemplifică utilizarea primitivelor fără blocare. Programul client foloseşte o variantă de emisie care se poate încadra în tipul (1) de primitive *send* fără blocare (*no wait send*), iar programul receptor foloseşte o variantă de recepţie analoga variantei (3) de mai sus. Recepţia unui mesaj este semnalată printr-o întrerupere; la generarea întreruperii, se lansează în execuţie funcţia declansată ca Event-Syncronous-Routine (ESR), care are rolul de a prelua şi memora mesajul într-un buffer liber. Serverul preia succesiv cate un mesaj din buffere, eliberează buferrul respectiv, tratează cererea şi trimite răspunsul înapoi la client; în tot acest timp, dacă soseşte o altă cerere, ea este memorată pentru o tratare ulterioară.

3.2.3 Primitive cu şi fără bufferare

S-a aratat în sectiunea precedentă că, în cazul primitivelor fără blocare este necesara mentinerea unor buffere, atît la nivelul clienţilor, cît şi la nivelul serverelor, pentru a permite memorarea mai multor cereri inca netrimise, respectiv inca nereceptionate. Utilizarea bufferelor ridică probleme specifice: alocare, eliberare, acces în regim de excludere reciproca. O modalitate des intalnita în cadrul sistemelor de operare distribuite de a rezolva problema bufferării mesajelor este de a defini o structura de date de tip *cutie postală sau port*, la care se trimit toate mesajele apartinînd unui destinatar; procesul destinatar crează cutia postală printr-o primitiva pusă la dispoziţie de nucleu. Problema spatiului limitat nu este însă eliminată nici prin acest procedeu (cel mult apare mai rar situatia bufferelor pline); ca atare, emitatorul trebuie blocat la o operaţie *send*, pînă la primirea unui mesaj de confirmare că mesajul a fost preluat.

Exemplul din anexa 1 arată o modalitate de bufferare la nivelul serverului. Dacă numărul de clienţi este foarte mare, s-ar putea sa se depaseasca capacitatea de memorare a bufferelor; în acest caz, clienţii repeta cererea, dupa o mică întîrziere, pentru a nu sufoca serverul.

Intr-un sistem care nu foloseşte bufferarea, procesele se sincronizează în momentul transferului. Aceasta strategie este o extensie a problemei rendez-vous, cunoscuta din sistemele centralizate. Semantica pentru rendez-vous între procese aflate pe noduri diferite (la distanţa) este aceeaşi ca şi în procesele locale (fig.



a) Serverul ajunge primul la rendez-vous *a) Clientul ajunge primul la rendez-vous*

Fig. 3.2.3.1. Mecanismul de rendez-vous

BUPT

3.2.3.1). Strategia este adevarată pentru modelul client/server şi presupune circulaţia informaţiei în ambele sensuri.

Pentru ca un client sa declanşeze rendez-vous cu un server, sunt necesare următoarele:
(1) clientul trebuie să cunoască definiţia interfeţei serverului;
(2) clientul trebuie să cunoască localizarea serverului: identificatorul procesului şi adresa nodului. Identificatorul procesului server poate fi determinat, avînd în vedere urmatoarele considerente:
(a) în fiecare nod se execută un server de nume, al cărui identificator este cunoscut; orice server este inregistrat de serverul de nume local, printr-un nume şi informaţiile de localizare;
(b) serverele de nume pot comunica între ele, tot prin rendez-vous; în acest mod, orice client poate determina localizarea unui server de pe orice nod.

3.2.4. Primitive sigure şi nesigure

În cadrul unei comunicaţii interprocese sigure, la execuţia primitivelor *send*, subsistemul de comunicaţie rezolva problema mesajelor pierdute pe baza confirmărilor şi retransmisiilor, iar la execuţia primitivelor *receive* se transmit confirmări. Aceasta înseamnă că în urma unei operaţii *send* sigure, terminate cu succes, procesul emitator este sigur că mesajul a fost receptionat (fiind confirmat). Comunicatiile folosind primitive nesigure, care nu garantează livrarea mesajelor, pot fi facute însă sigure prin translatarea intrgului mecanism de confirmare şi retransmisii la nivelul proceselor. Unii autori [Mul89] chiar sugerează plasarea mecanismelor de recuperare a erorilor la nivel inalt). Pentru ambele soluţii de recuperare a erorilor, există două tipuri de semantică posibile, definite; în funcţie de actiunea pe care o întreprinde un server, la primirea unei cereri duplicate: (1) semantica *cel-puţin-o-data*, prin care se asigură că o cerere a unui client este tratată cel puţin o dată (fig. 3.2.4.1); (2) semantica *exact-o-data*, prin care se asigură execuţia numai o dată a codului de tratare a unei cereri; aceasta este absolut necesară în cazurile în care tratarea aceleiaşi cereri de mai multe ori conduce la erori (cererile a caror tratare se poate repeta fără prejudicii se numesc *idempotente*; un exemplu de cerere neidempotentă este o operaţie *append* într-un fişier).

Pentru a se implementa primitive de tipul (2), serverele trebuie sa menţină o tabelă cu identificatorii cererilor tratate şi raspunsurile trimise clienţilor pentru aceste cereri; la primirea unei cereri duplicate, serverul nu mai execută codul de tratare a cererii respective, ci doar transmite rezultatul.

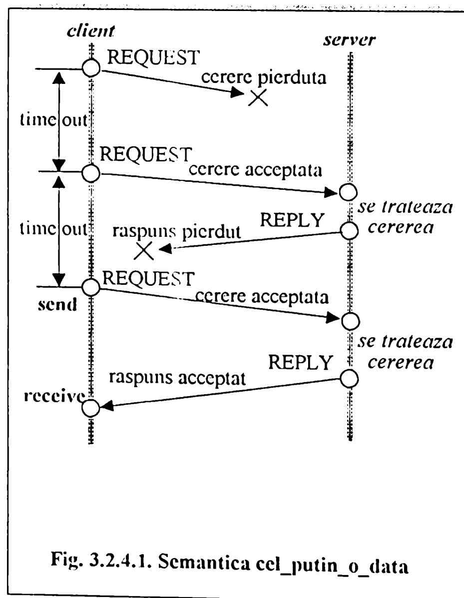


Fig. 3.2.4.1. Semantica cel_putin_o_data

BUPT

3.2.5 Forme structurate ale comunicării prin mesaje

Pentru a se obține performanțe superioare în comunicația prin mesaje, aplicată modelului client-server, ținînd cont că un mesaj REQUEST al unui client este urmat de un mesaj REPLY, se pot combina primitive *send* și *receive* de la nivelul clientului într-o singură primitiva *send*. Acest lucru simplifică bufferarea mesajelor, deoarece cererea poate ramîne în bufferul clientului, iar raspunsul serverului poate fi memorat în acest buffer și protocolul, deoarece controlul erorilor și al fluxului interpretează răspunsul la o cerere ca și o confirmare. Setul de primitive folosit este reprezentat în fig. 3.2.5.1.

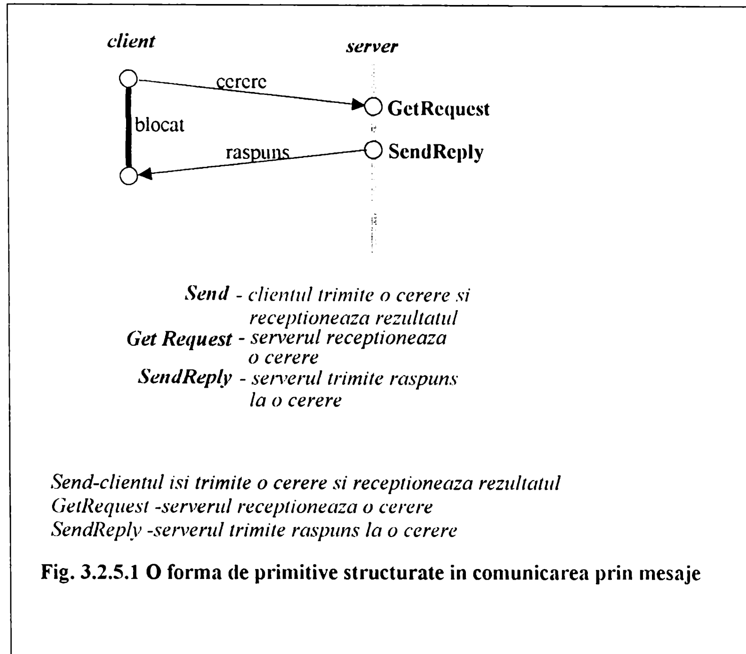


Fig. 3.2.5.1 O forma de primitive structurate in comunicarea prin mesaje

3.2.6. Adresare directa și indirecta

Comunicarea între procese prin mesaje dispune de două tehnici de adresare: *adresarea directă*, în cadrul căruia emițătorul specifică explicit procesul destinatar și *adresarea indirectă* în cadrul căruia emițătorul specifică drept destinație un nume global, denumit *cutie poștală* sau uneori *port*.

Adresarea directa în cele două variante, *simetrică* și *asimetrică* (fig. 3.2.6.1.a,b) are următoarele caracteristici: (1) între două procese care doresc să comunice, și care trebuie sa-și cunoască reciproc identitatea, se stabilește automat o cale de comunicare; (2) între două procese care comunică, există numai o cale de comunicație; (3) comunicația este bidirecționala. *Adresarea directa* prezintă avantaje precum o implementare simplă și posibilitatea controlului momentelor de transmisie a mesajelor. În schimb, modificarea numelui unui proces impune modificare în alte procese a tuturor referințelor la acesta; în plus, adresarea directă nu permite trimiterea unei cereri la mai multe servere identice. O variantă a adresării directe adoptată în două sisteme de operare distribuite experimentale ([Gos91]). DEMOS/MP și Charlotte este *comunicarea prin legături*. O *legătură* este definită ca un canal de comunicație unidirecțional sau bidirecțional la un proces utilizator, proces de sistem sau nucleu. Fiecărui proces i se atasează o tabela de legaturi, care identifică în fiecare intrare o linie de comunicație; procesul poate crea, distruge legaturi și transfera un capat al unei legaturi la alt proces (fig. 3.2.6.1.). O legatură poate fi privita ca o conexiune, într-un protocol orientat pe conexiuni, în care se poate lucra cu sau fără blocare, cu sau fără bufferare de mesaje.

În cadrul adresării *indirecte*, o cutie poștală poate apare ca destinație în orice primitiva *send*, sau ca sursa în orice primitiva *receive*. Deși o astfel de comunicație este adecvată pentru modelul client-server, implementatrea ei fără un suport specializat în rețea (un mesaj trebuie transmis de subsistemul de comunicație către mai multe noduri) poate fi neperformantă. Ca atare, în sistemele de operare distribuite se implementează,

BUPT

Send (P_process, mesaj);# se trimite un mesaj procesului P.
Receive (Q_process, buffer);# se receptioneaza un mesaj de la procesul Q
a) Adresare directa simetrica

Send (P_process, mesaj);# se trimite un mesaj procesului P.
Receive(id, buffer);# se receptioneaza un mesaj de la orice proces; id preia numele procesului emitator
b) Adresare directa nesimetrica

Send (link, mesaj);# se trimite un mesaj pe legatura link.
Receive(link, buffer);# se receptioneaza un mesaj din legatura link.
Send (link, buffer, link_end);# se transfera un capat al unei legaturi pe legatura link si se depune in buffer
c) Adresare directa cu legaturi

Send (P_port, mesaj);# se transmite un mesaj in portul P.
Receive (P_port, mesaj);# se receptioneaza un mesaj din portul P.
Send (P_port.port);# se transporta dreptul de proprietate la un port
d) Adresare indirecta

Fig. 3.2.6.1. Primitive pentru adresarea directa si indirecta.

în general, un tip simplificat particular de cutie poştală numit *port*. Un *port* este un obiect al nucleului, protejat, catre care procesele pot să transmită, respectiv extrage mesaje. Comunicarea prin porturi are urmatoarele caracteristici: (1) asociază o cale de comunicare la mai mult de două procese; (2) între două procese se pot stabili mai multe căi de comunicare, fiecare corespunzînd unui port; (3) comunicaţia poate fi fie unidirecţionala, fie bidirecţionala. Accesul pentru extragerea mesajelor din porturi a proceselor poate fi condiţionat prin drepturi de recepţie (de exemplu, prin obţinerea unei capabilitati). Porturile trebuie declarate; declararea unui port înseamnă asocierea la port a unei cozi de mesaje, gestionată în mod normal printr-un algoritm FIFO (ordinea FIFO poate fi modificată prin mesaje urgente, care se tratează în mod special). Porturile pot avea ca proprietar fie un proces (1), fie sistemul de operare (2). Cazul (1) implică urmatoarele: (a) creatorul portului este şi proprietarul; el poate extrage mesaje din port, spre deosebire de celelalte procese care pot doar trimite mesaje la acest port; (b) portul este ataşat procesului proprietar, astfel încît distrugerea acestui proces înseamnă şi distrugerea portului; oricăror alte procese, care transmit în continuare mesaje la acest port, va trebui să li se semnaleze o eroare (eventul printr-un mecanism de tratare excepţii); (c) există 2 modalităţi de stabilire a proprietarului şi utilizatorilor unui port: fie se permite unui procesor, care devine astfel proprietar, declararea unei variabile port, fie se declară un port partajat şi apoi se stabileşte proprietarul; (d) proprietarul unui port poate, printr-o primitivă, transmite altui proces drepturile de proprietate (fig. 3.4.6.1.d). Cazul (2) presupune că sistemul de operare pune la dispoziţie primitive pentru crearea unui port, (procesul apelant este şi proprietarul portului) distrugerea unui port şi trimiterea şi recepţia mesajelor. În cazul în care un proces încearcă să trimită mesaje într-un port în care nu mai există loc liber, există trei soluţii: (1) procesul este blocat pînă la plasarea mesajului; (2) procesului i se semnalează o eroare; (3) mesajul este acceptat de nucleu şi trimis de acesta atunci cînd mesajul poate fi plasat.

Porturile pot fi *statice* (porturi prin care o pereche de procese comunică pe tot parcursul duratei de viata) sau *dinamice* (porturi create, asociate şi utilizate pentru o perioada de timp). Porturile statice sunt adecvate pentru o interacţiune între procese de tipul corutinelor (fig. 3.2.6.2.) sau proceselor distribuite care necesita partajarea unor informaţii împreună cu un anumit nivel de sincronizare.

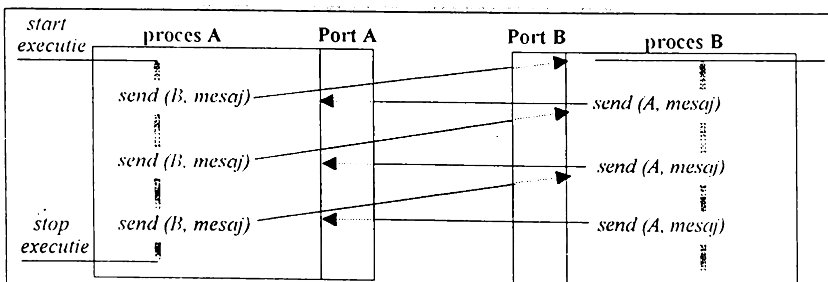


Fig. 3.2.6.2. Comunicarea prin porturi folosita pentru a obtine o interactiune de tip corutine.

BUPT

3.2.7. Excepţii

În aceasta secţiune se iau în considerare probleme legate de terminarea procesului emiţător sau receptor sau de pierderea mesajelor în reţea. Dacă procesul receptor se termină fără ca emiţătorul sa cunoască acest lucru, emiţătorul va continua sa transmită mesaje. Pentru primitivele cu bufferare, execuţia continuă, eventual pînă la depăşirea capacităţii bufferelor. Dacă emiţătorul trebuie sa ştie dacă mesajul a fost preluat de receptor, atunci trebuie folosit un protocol cu confirmări. Cînd se foloseşte o primitiva *send* fără bufferare, pentru ca emiţătorul să nu se blocheze la infinit, sistemul de operare trebuie fie sa termine emiţătorul, fie sa semnaleze că receptorul s-a terminat. Analog trebuie procedat în cazul cînd un server este blocat în aşteptarea uñui mesaj de la un client care s-a terminat. Un exemplu în acest sens este scrierea de catre un proces într-un soclu, la capatul căruia procesul server este terminat, în sistemele UNIX. În acest caz, sistemul de operare generează pentru procesul client semnalul SIGPIPE a cărui tratare implicita este terminarea procesului.

În ceea ce priveste pierderea mesajelor, în sectiunea 3.1.3. s-a arătat că prin mecanismul timeout din protocolul software acestea pot fi depistate. Problema care se ridică aici este la care nivel se face recuperarea erorilor. Există 3 soluţii: (1) sistemul de operare este responsabil pentru detecţie şi pentru transmisie; (2)procesul emiţător este responsabil pentru detecţie şi retransmisie; (3) sistemul de operare este responsabil pentru detectie, el semnalează emiţătorului că mesajul a fost pierdut şi retransmisia este lăsată pe seama procesului emiţător (dacă aceasta o doreşte). În general pentru alegerea unei soluţii trebuie luate în considerare următoarele observatii: (1) actiunea depinde şi de specificul aplicaţiei; (2) erorile necritice pot fi ignorate; cele care duc însă la distrugerea unei componente a sistemului trebuie tratate; (3) în cazurile în care o eroare dintr-un server are efecte globale, trebuie aleasa o comunicare prin tranzactii atomice (vezi capitolul 7) .

3.2.8 Exemplu. Comunicaţie prin porturi în Mach

Arhitectura sistemului distribuit Mach, creat la Carnegie-Mellon de o echipa de cercetatori condusi de Richard Rashid ([Ras88] [Ras81]) este prezentată în fig. 3.2.8.1.

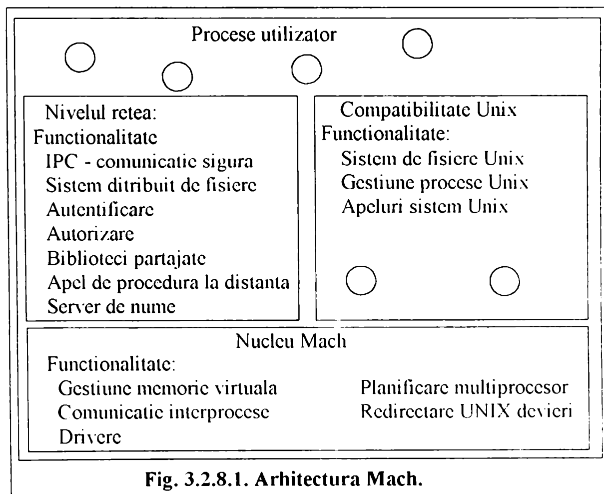


Fig. 3.2.8.1. Arhitectura Mach.

Comunicatia interprocese în Mach se bazează în întregime pe *porturi şi mesaje*. Un *port* este definit ca un canal de comunicaţie, implementat printr-o coada de mesaje protejată de nucleu, de lungime finita. În Mach, porturile sunt referintele pentru obiecte: porturile reprezinta obiectele; Mach este proiectat utilizînd modelul obiect (vezi subcapitolul 1.4.2). Un port poate avea mai mulţi emiţători, dar un singur receptor. Drepturile de acces la un port constau în acces pentru citire, scriere, şi dreptul de proprietar. Cind intr-un task se creează un nou port, taskul primeşte toate cele trei drepturi de acces la port; un fir al taskului nu poate referi decît un port la care taskul respectiv are dreptul de acces. Un task poate transmite drepturile de acces la un port altui task, utilizînd mesaje care conţin capabilitati pentru port. Dacă un task transmite dreptul de recepţie la un port, el pierde acest drept (dreptul de emisie la un port nu se pierde); în perioada de transmisie a dreptului de receptie, toate mesajele transmise la acel port sunt reţinute de nucleu, care le transmite apoi noului task ce a primit dreptul de receptie. Un *mesaj* în Mach este obiect de date, utilizat pentru comunicaţie între fire. Mesajele nu sunt limitate ca dimensiune, dar au header de lungime fixa care conţine: *msg size*, dimensiunea mesajului de transmis sau dimensiunea maximă a mesajului de receptionat, *msg local port*,

BUPT

msg_remote_port, numele porturilor locale sau la distanţă la/de la care mesajul trebuie transmis/ receptionat, *msg_id* un identificator al tipului mesajului; în partea de lungime variabilă, mesajele pot conţine şi pointeri sau capabilităţi pentru porturi. Primitivele de lucru cu porturile sunt urmatoarele: (1) *msg_send(mesaj, optiune, timeout)* pentru transmisia unui mesaj la portul specificat în headerul mesajului; argumentul *timeout* specifică timpul maxim de aşteptare pentru cazul în care portul destinaţie este plin cu mesaje. Receptorul poate utiliza *msg_local_port* pentru a trimite un raspuns emiţătorului (cu excepţia cazului cînd *msg_local_port* este PORT_NULL); (2) *msg_receive(mesaj, optiune, timeout)* pentru recepţia unui mesaj de la portul specificat în headerul mesajului, *msg_local_port* sau în grupul de porturi implicite; *timeout* este intervalul maxim de aşteptare a sosirii unui mesaj; (3) *msg_rpc(mesaj, optiune, rcv_dim, send_timeout, receive_timeout)* pentru trimiterea unui mesaj, urmat de receptia unui raspuns, utilizînd acelaşi buffer; *msg_remote_port* specifică portul la care mesajul trebuie transmis, iar *msg_local_port* specifică portul la care mesajul este receptionat; *send_timeout* şi *receive_timeout* sunt timpii de aşteptare la emisie, respectiv la receptie. Pentru toate primitivele (1), (2), (3) *optiune* specifică conditiile de eroare în care primitivele trebuie sa fie terminate.

Primitiva *msg_rpc()* oferă facilitati de apel de procedură la distanţa prin intermediul porturilor.

Pentru manipularea drepturilor de acces la porturi şi receptia mesajelor de control, Mach mai prevede urmatoarele primitive: (1) *port_allocate(task, port)* crearea unui port, obţinandu-se toate cele trei tipuri de drepturi; identificatorul portului se returnează în port; (2) *port_deallocate(task, port)* distruge drepturile de acces la port; dacă taskul era proprietar, portul este distrus şi se anunta toate taskurile cu drept de emisie la acest port; dacă taskul avea dreptul de receptie, el pierde acest drept, care se transmite proprietarului; (3) *port_enable(task, port)* adauga acest port la grupul de porturi implicite pentru *msg_receive;* (4) *port_disable(task, port)* elimină acest port din grupul de porturi implicite; (5) *port_messages(task, porturi, contor_porturi)* returnează pentru un task specificat, un tablou de porturi active cu mesaje în aşteptare; (6) *port_status(task, port, num_msg, backlog, receptor, proprietar)* returnează informaţii de stare relative la *port:* numărul de mesaje în aşteptare *(num_msg)*, numărul de mesaje care mai pot fi primite fără a se bloca emiţătorul *(backlog)*, indicatori ai drepturilor de acces, pentru recepţie (*receptor*), respectiv dreptul de proprietar (*proprietar*); (7) *port_backlog(task, port, backlog)* returnează numărul de mesaje care mai pot fi trimise fără a bloca emiţătorul la acest port.

În Mach, mesajele pot fi trimise sincron sau asincron; în ultimul caz, receptionarea unui mesaj se notifică printr-un semnal Unix; se pot folosi însă şi fire pentru tratarea operaţiilor de recepţie asincrone. Pentru transmiterea mesajelor la distanţa se folosesc serverele de reţea astfel încît se asigură transparenţa schimbului de mesaje între procese (fig.3.2.8.2). Nucleul Mach nu recunoaşte o transmisie la distanţă; din punctul de vedere al nucleului, toate comunicaţiile interprocese se fac local. Serverele de reţea, în afară rolului extinderii mecanismului de comunicaţie la procesele din reţea, asigură şi siguranţa transmisiei şi conversia datelor la transmisie.

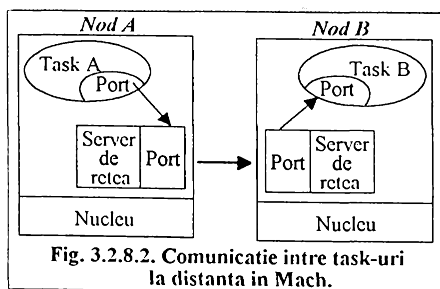


Fig. 3.2.8.2. Comunicatie intre task-uri la distanta in Mach.

BUPT

3.3. Comunicarea între procese prin conducte

Comunicarea între procese prin conducte (*pipes*) este un mecanism care permite transferul datelor între procese utilizînd strategia FIFO, concomitent cu sincronizarea proceselor implicate în transfer. Ea poate fi aplicată şi modelului client server prezentat anterior deşi nu îi este caracteristică.

3.3.1. Prezentarea generală a comunicaţiei prin conducte

În mod traditional, conductele au fost implementate utilizînd sistemul de gestiune al fişierelor. Proprietăţile principale ale conductelor sunt acelea ca: (1) permit schimbul de informaţie între procese care nu-şi cunosc identitatea; (2) permit transmiterea datelor în avalansa. (3)Iesirea unuia sau mai multor procese poate fi conectată la intrarea unuia sau mai multor procese.

În Unix o conductă apare ca un flux de octeţi; nu există mecanism de transmisie out-of-band iar bufferul ei este de capacitate limitată. Cînd un proces încearcă sa citească date dintr-o conductă vida, se blochează pînă cînd conducta este completată cu volumul de date necesar. Un proces care încearca sa citeasca date dintr-o conductă la capatul căreia nu mai există nici un proces care scrie primeşte indicatia end-of-file; în schimb dacă un proces încearca sa scrie într-o conductă la capatul căreia nu există cititori se genereazā semnalul SIGPIPE. Un proces care aşteaptă pentru citire dintr-o conductă, sau altul care aşteaptă pentru scriere (cînd bufferul conductei este plin) poate fi întrerupt. Există două tipuri de conducte: *conducte fără nume* şi *cu nume*; diferenţa între cele două este dată de modul în care sunt create şi accesate initial. Astfel, în sistemul UNIX conductele fără nume se creează cu apelul sistem *pipe* în timp ce conductele cu nume se creează cu apelul *mknod*; conductele fără nume nu pot fi accesate decît din procesele descendente ale procesului care le-a creat. În ambele cazuri, accesul la conducte se face cu apelurile *read, write*. O conductă cu nume este luată în evidenţă de sistemul de gestiune al fişierelor; unul sau mai multe procese pot să deschidă conducta specificînd modul *read*, în timp ce altele pot specifică la deschidere modul *write*. Dupa deschidere, o conductă cu nume se comporta exact ca şi o conductă fără nume.

Pentru comunicaţia la distanţa prin pipe sistemul de operare distribuit LOCUS a extins ambele tipuri de conducte.

3.3.2 Comunicarea prin conducte în Microsoft WIN32

Microsoft WIN32 furnizează pentru comunicaţia la distanţa *conducte cu nume*, pentru a căror implementare utilizează servicii orientate pe conexiune; ca atare trebuie sa existe un procedeu pentru ca aplicaţiile server sa aştepte cererile de conectare. Acest procedeu consta în: (1) *crearea unei conducte* cu nume, executînd apelul *CreatNamedPipe()*; (2) aşteptarea cererilor de conectare, dupa execuţia apelului *ConnectNamedPipe()*. Conductele au cateva caracteristici specifice care stabilesc cu apelul *CreatNamedPipe()*: (1) *directia*. O conductă poate fi *bidirecţională sau unidirecţională* client-la-server sau servre-la-client; (2) *modul de scriere*. În conducte se pot scrie *mesaje* (modul *mesaj*) sau *octeti* (modul *byte*) şi se pot citi mesaje şi octeti. (3) *modul citire*. Pentru o conductă din care se citeşte în modul byte, operaţiile de scriere şi citire nu trebuie sa manipuleze simultan acelaşi număr de octeti; în schimb, pentru o conductă mod mesaj, fiecare scriere în conductă defineşte un mesaj, astfel că citirile trebuie efectuate la nivel de mesaje. Incercarea de a citi un număr mai mare de date decît un mesaj, are ca efect returnarea numai a mesajului; în schimb încercarea de a citi parţial un mesaj returnează eroare; (4) *modul de blocare*. Există două moduri de lucru, cu blocare sau fără; ele se referă la blocarea apelantului unei funcţii ConnectNamedPipe(). Modul fără blocare este rar folosit, fiind prevazut pentru compatibilitate cu conductele din OS/2. În schimb, ca şi fişierele standard, conductele pot fi deschise pentru operaţii de I/O asincrone (overlapped).

Pentru contactarea unui server printr-o conductă, un client trebuie sa execute apelul *CreatFile()*, care deschide conducta cu nume, (numele fiind scris sub o formă speciala: \\<nume_nod>\PIPE\ <nume_conducta>) implicit în modul cu blocare şi citire la nivel de octet. Pentru a modifică aceste setări implicite, se poate utiliza funcţia *SetNamedPipeHandleState()*. Dupa deschiderea conductei, clientul şi serverul pot utiliza pentru schimbul de date funcţiile *ReadFile()* şi *WriteFile()* (diferentele faţă de utilizarea unui fişier şi o conductă fiind în acest caz transparente) sau funcţia *TransactNamedPipe()*, care permite scrierea unui mesaj într-o conductă şi aşteptarea unui raspuns (utila pentru modelul client server).

Serverele pot fi proiectate sa funcţioneze *multifir*. În acest sens, o soluţie este sa se creeze mai multe instanţe ale unei conducte, iar fiecărei instanţe să i se asocieze un fir în cadrul căruia se aşteaptă cereri de conectare şi se deservesc cereri de servicii. Schema acestui mod de lucru este prezentată în figura 3.3.2.1.

BUPT

```
const short NO_PIPES=3;
HANDLE hPipeInstance[NO_PIPES], hPipeThread[NO_PIPES];
for (int i=0;i<NO_PIPES;++i) {
  hPipeInstance[i] = CreatNamedPipe (...);
  if (hPipeInstance[i] != INVALID_HANDLE_VALUE) {
    hPipeThread[i] = CreatThread (NULL, 0, NamedPipeThread, hPipeInstance[i], ...);
  }
}
DWORD WINAPI NamedPipeThread (HANDLE hPipeInstance) {
  DWORD dwError;
  BYTE byBuffer[4096];
  DWORD dwBytes;
  while (ConnectNamedPipe (hPipeInstance, NULL) || GetLastError() == ERROR_PIPE_CONNECTED)
    while (ReadFile (hPipeInstance, byBuffer, sizeof(byBuffer), &dwBytes, NULL)) {
      //trateaza cererea
    }
    DisconnectNamedPipe (hPipeInstance);
  }
  ExitThread (dwError = GetLastError());
  return dwError;
}
```

Fig. 3.3.2.1. Utilizarea conductelor in servere multifir

3.4. Modelul apelului de procedură la distanţa

Modelul de comunicaţie bazat pe *apel de procedură la distanţa (RPC=Remote Procedure Call)* permite unui proces să apeleze şi să execute o procedură de la distanţă ca şi pe o procedură locală. Pe timpul execuţiei procedurii de la distanţă, apelantul este blocat, pînă cînd se receptionează rezultatul. Prin RPC se permite astfel extinderea apelului de proceduri, folosit în limbajele de programare la mediile distribuite. Modelul RPC este atractiv pentru că oferă un mecanism de comunicaţie în cadrul programelor distribuite, care se poate caracteriza astfel: (1) este simplu şi familiar (datorita similarităţii cu apelurile de proceduri locale); (2) este general (apelul de procedură este folosit în toate aplicaţiile); (3) poate fi implementat eficient.

Modelul RPC simplifică construcţia programelor distribuite, permiţînd ignorarea de catre acestea a detaliilor de comunicaţie şi a erorilor de transmisie.

Modelul RPC a fost utilizat în multe sisteme distribuite începînd cu Argus ([LS83]), HRPC (University of Washington, [BC85]), Eden [ABL85], Spice (Carnegie Melon University, [JRT85]) şi terminînd cu variante pentru sisteme Unix, precum Xerox Courier RPC, creat la Xerox Parc pentru a fi utilizat în mediile MESA şi Cedar [BN84], SUN RPC (SUN Microsystems [Sun85]), V Kernel creat de Cheriton la Stanford University, Apollo RPC (Hewlett Packard) şi mai recentele OSF DCE (Open Software Fondation Distributed Computing Environment) şi Microsoft Windows NT şi Windows '95. Printre acestea, Argus prezinta o caracteristică specială: aceea că integrează modelul RPC cu mecanismul tranzactiilor atomice. Unele sisteme din cele de mai sus (Argus, Cedar) au fost initial proiectate pentru a lucra cu un singur limbaj, în timp ce altele au fost destinate pentru lucrul cu diverse limbaje şi chiar în medii eterogene.

Desi simplitatea modelului RPC îl recomandă ca o paradigmă principală pentru comunicaţia în aplicaţii distribuite, performantele oferite de acesta pot sa nu fie totuşi adecvate pentru unele aplicaţii. Aceasta deoarece, sistemele RPC, în majoritate , sunt proiectate sa furnizeze un raspuns cît mai rapid unei cereri, astfel încît apelurile individuale sunt deservite relativ repede, dar productivitatea este mica. Pentru aplicaţiile în care se cere o productivitate mare de deservire a cererilor, rămanînd pe planul doi rapiditatea raspunsului, modelul RPC s-ar putea sa nu ofere performanţele dorite.

BUPT

3.4.1. Prezentarea generală a modelului

Ideal este ca un apel de procedură la distanţă să aibă aceeaşi semantică şi sintaxa ca şi un apel de procedură locală. În practică însă, acest ideal este dificil de realizat datorită următoarelor cauze: (1) erorile de transmisie sunt greu de mascat complet; (2) procesele au spaţii de adrese diferite; (3) procesele au durate de viaţă diferite.

Modelul RPC trebuie sa fie dotat cu un protocol de transport, care sa furnizeze transmisia parametrilor procedurii ca şi a rezultatului; uneori chiar termenul de RPC este folosit pentru a desemna acest protocol de transport.

Comunicaţia prin mesaje şi extensiile sale (formele structurate de mesaje) furnizează servicii rareori situate deasupra nivelului de transport al modelului ISO/OSI. Aceste funcţii sunt în general încapsulate în programele de aplicaţii. Spre deosebire de comunicaţia prin mesaje, protocolul RPC este responsabil şi de maparea caracteristicilor de limbaj ale unui apel de procedură în comunicaţiile oferite de nivelul transport al reţelei.

Desi comunicaţia prin mesaje şi apelul de procedură la distanţă sunt analoge (în sectiunea 3.2.2. fig. 3.2.2.1.. se arată patru moduri de interacţiune a proceselor de la distanţa, dupa modul de returnare a controlului apelantului, apelul de procedură fiind clasificat ca al 4-lea mod), ele diferă prin urmatoarele: (1) în timp ce la comunicaţia prin mesaje, procesele care interacţionează pot fi vazute ca partenere, la RPC relatia dintre ele este de tip master-slave; (2) în comunicaţia prin mesaje, procesele completează explicit cîmpurile unui mesaj înainte de transmisie, ca şi despachetarea rezultatelor dintr-un mesaj.

Andrews şi Schneider [AS83], identifică două moduri de a specifica un serviciu al unui server într-un mecanism RPC (fig. 3.4.1.1.):

(1) serviciul se declară ca o procedură dintr-un limbaj secvential.

```
remote procedure service ( in parametrii_cu_valoare;
                           out parametrii_rezultat)
        corp
    end
a)declarare ca o procedura
    accept service ( in parametrii_cu_valoare;
        out parametrii_rezultat) -> corp
b)declarare ca o instructiune
```

Fig. 3.4.1.1. Două moduri de specificare a serviciilor în modelul RPC

În acest caz procedura se implementează într-un proces server, acesta aşteaptă sosirea unui mesaj continînd valorile argumentelor de la un proces client, asignează aceste valori parametrilor procedurii, execută procedura, şi returnează un mesaj de raspuns continînd valorile parametrilor rezultat. Procesul client va fi blocat pe parcursul execuţiei procedurii chiar dacă nu există parametrii rezultat.

(2) serviciul se declara ca o instructiune, care poate fi plasată oriunde în procesul server. Interacţiunea între procesul server şi client este de tipul rendez-vous.

În continuarea acestui capitol se va considera modelul RPC numai pentru primul mod.

Implementarea modelului RPC este prezentată sub o formă generală în fig. 3.4.1.2. Se observa ca sunt necesare cinci elemente pentru a se obţine efectul execuţiei unei proceduri locale într-un mediu distribuit: *procesul client, codul stub pentru client, mecanismul de transport, codul stub pentru server şi procesul server*. Mecanismul de transport implementează transmisia sigură de mesaje; acesta poate utiliza protocoale de la nivelul transport, însă tendinţa implementarilor actuale este de a utiliza un protocol (neorientat pe conexiune) bazat pe datagrame scris special pentru RPC, care sa asigure o transmisie sigură a mesajelor. Modulele stub au ca scop impachetarea şi despachetarea parametrilor in mesaje.

Modelul RPC poate fi interpretat şi ca o extindere a primitivelor sigure, cu blocare, structurate *Send, GetRequest, SendResponse*. Spre deosebire de aceste primitive, datorita simplităţii şi totodată generalităţii sale, modelul RPC poate fi insă extins la mai multi clienţi şi la mai multe servere, ca şi la medii eterogene.

BUPT

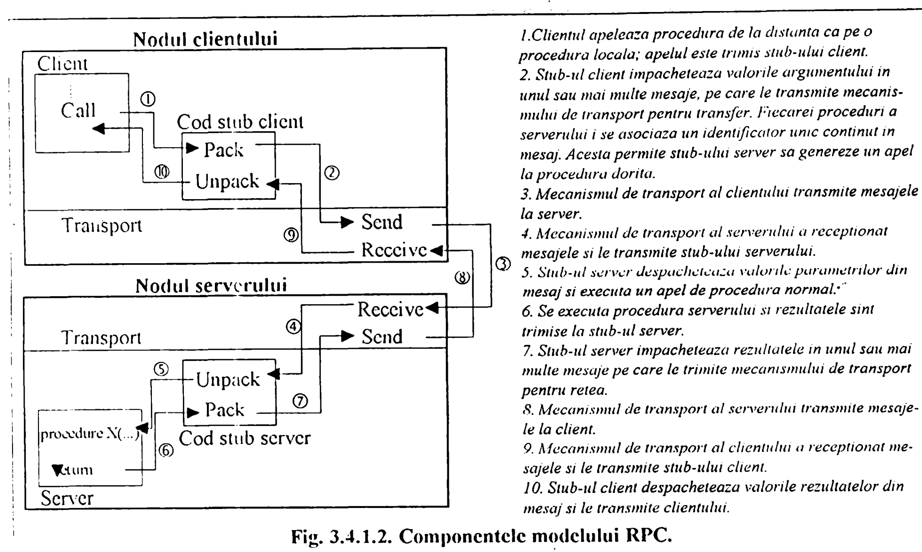


Fig. 3.4.1.2. Componentele modelului RPC.

3.4.2. Transmiterea parametrilor şi rezultatelor

O problema cheie a modelului RPC este transmiterea parametrilor şi reprezentarea parametrilor şi rezultatelor în mesaje.

Parametrii pot fi transmisi prin *valoare* sau *referinţa*; dacă transmiterea prin valoare nu pune nici o problema (valorile parametrilor se copiază direct în mesaj), în schimb transmiterea parametrilor prin referinţa şi a pointerilor ridică probleme. O soluţie este de a înlocui transmiterea parametrilor prin referinţa prin transmitere prin copiere; dacă parametrul este doar intrare pentru server, atunci nu mai este necesară copierea lui la loc în client iar dacă parametrul este doar ieşire nu este necesară transmiterea valorii lui la server.

Ca atare, se asociază fiecărei declaraţii a unei proceduri o specificaţie formală numita *interfaţă* (prin care se precizează tipul parametrilor, dacă sunt de intrare sau de ieşire (sau ambele) şi dimensiunea lor (maximă); această specificaţie este folosită de un compilator special care generează codul stub, stub-ul se linkeditează împreună cu clientul, respectiv serverul, pentru a avea acces la spaţiul lor de adrese. În fig. 3.4.2.1 se arată ca exemplu generarea programelor client şi server în SUN RPC.

Definirea interfeţei procedurilor serverului are mai multe scopuri: (1) primul este cel arătat mai sus, de a permite serverului (care exportă procedurile) şi clientului (care importă) să se pună de acord asupra parametrilor de intrare şi de ieşire. Impachetarea şi despachetarea parametrilor în/din mesaje ca şi dispecerizarea apelurilor la proceduri în stub-ul server se va genera pe baza definiţiei interfeţei; de aceea interfaţa contine caracteristici ale procedurilor furnizate de server care trebuie sa fie vizibile clientului (numele procedurilor şi tipurile parametrilor lor). Unele sisteme permit procedurilor apelate la distanţa sa semnaleze exceptii; aceste tipuri de exceptii trebuie documentate şi ele uneori în descrierea interfeţei. Mecanismele RPC includ, în general, şi un limbaj de interfata, IDL (Interface Description Language) pentru a permite programatorilor sa definească interfata. Un astfel de limbaj de interfaţă furnizează posibilitatea de a defini tipuri scalare (char, boolean, int, real), facilităţi de definire a tipurilor structurate (tablouri, stringuri, inregistrari), facilităţi pentru utilizarea pointerilor. Definirea interfeţei într-un astfel de limbaj este procesată de un compilator special. (2) al doilea scop este că declararea interfeţei serveşte ca bază pentru verificarea tipurilor parametrilor la compilare.; (3) reprezentarea datelor la cele două capete poate fi diferită, declaraţia interfeţei poate fi folosită ca bază pentru generarea conversiei de cod necesară.

Problema conversiei între reprezentari diferite ale datelor la cele două capete poate fi rezolvată în două moduri: (a) prin definirea unei reprezentari standard pentru fiecare tip de date (din mesaje) şi conversia mesajelor, în fiecare capat, la reprezentarea standard respectiva. Aceasta soluţie oferă portabilitate între diverse sisteme şi permite programelor de aplicaţii ignorarea unor detalii de reprezentare a datelor precum ordinea octetilor (*big* sau *little endian*). (b) Clientul trimite mesajele utilizînd propria sa reprezentare a datelor (eventual cu cativa indicatori pentru identificarea reprezentării utilizate) iar serverul realizează conversiile necesare. Aceasta soluţie are dezavantajul că serverul trebuie sa poată accepta şi converti orice reprezentare particulară (problemele cresc cînd se adauga noi calculatoare sau limbaje în sistem), dar şi avantajul că nu este necesară nici o conversie în cazul comunicaţiei între două sisteme cu aceeaşi arhitectură.

BUPT

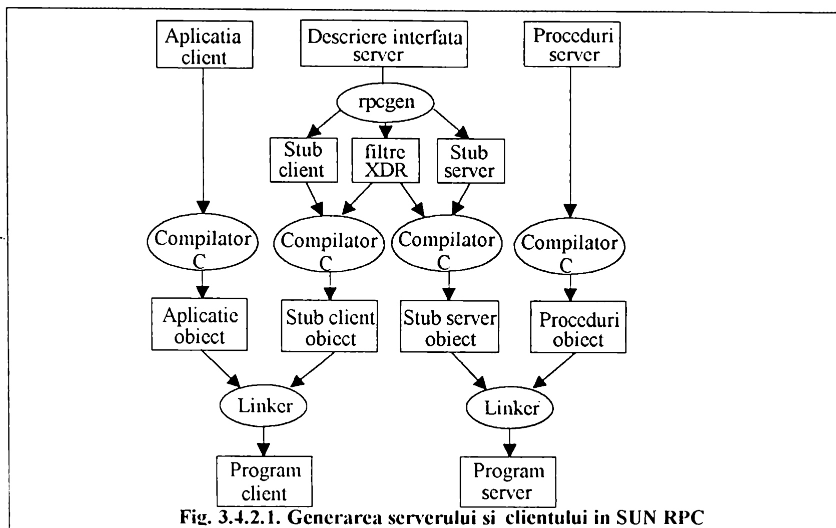


Fig. 3.4.2.1. Generarea serverului si clientului in SUN RPC

Ambele soluţii sunt utilizate. Astfel, SUN RPC şi Xerox Courier folosesc prima soluţie; SUN RPC defineşte reprezentarea standard XDR, care impune o ordine big-indian şi o dimensiune minimă pentru orice tip de 32 biţi (aceasta înseamnă că pentru a transmite un întreg de 16 biţi, se folosesc 32 biţi în ordinea big-endian) iar Xerox Courier RPC defineşte o reprezentare tot cu ordinea big-endian dar cu dimensiunea minimă de 16 biţi. Apollo RPC şi DCE RPC folosesc soluţia a doua. Apollo RPC permite mai multe formate (big sau little endian, formatele IEEE, VAX, IBM şi Cray pentru numere reale); emiţătorul transmite în formatul sau propriu (dacă este unul din formatele suportate) şi receptorul realizează conversia (soluţia a doua); DCE RPC permite, analog, mai multe formate şi face conversia tot la receptor (metoda se numeste *receiver makes right*). Ambele soluţii utilizează ceea ce se numeste *implicit typing*, adică se transmite valoarea unei variabile (într-un format standard) şi nu tipul acelei variabile, în contrast cu tehnica ISO OSI şi ASN.1, care transmite în mesaje tipul fiecărui camp de date (codificat pe 8 biţi) şi apoi valoarea acelui cîmp (*explicit typing*).

3.4.3. Legarea clienţilor la servere

Legarea clienţilor la servere se referă la contactarea acelui sistem de la distanţă care deţine procedura de executat. Există două componente ale procesului de legare: (1) localizarea calculatorului pentru serviciul dorit; (2) localizarea procesului server de pe acel calculator.

O primă tehnică impune ca fiecare client sa cunoască nodul care trebuie contactat; un superserver de pe acest nod, a cărui adresa este publică, înregistrează adresele (porturile) tuturor serverelor disponibile de pe nod. O altă tehnică este transmiterea unui mesaj broadcast (sau multicast) în vederea obţinerii localizării de la superservere. O tehnică mai generală este utilizarea unei baze de date centralizate, furnizată de o autoritate centrală în sistem (un server de nume denumit *binder)*. La începutul execuţiei sale, orice server va exporta interfaţa sa; procedura este denumita *înregistrarea serverului* în baza de date. Pentru înregistrare, serverul livrează numele său, un număr de versiune, un identificator unic (în general de 32 biţi) şi un *handle* pentru a fi localizat; handle-ul este o caracteristică a sistemului şi poate fi o adresa IP, o adresa Ethernet, un identificator de proces. Pot fi livrate şi alte informaţii, ca de exemplu legate de autorizarea clienţilor. Funcţia de înregistrare a serverului este pusă la dispozitia programelor de o bibliotecă de funcţii run-time; de asemenea, pot fi furnizate şi funcţii pentru deînregistrarea unui server, sau obţinerea handle-ului şi identificatorului unic al unui server ale cărui nume şi versiune se cunosc.

Legarea clientului la server se face de catre stub-ul client, cînd clientul cere pentru prima oara execuţia unei proceduri la distanţa; stubul client trimite un mesaj la binder cerînd importul interfeţei serverului X, versiunea Y (X, Y sunt cunoscute, stub-ul client fiind compilat din definiţia interfeţei serverului, un model al procesului de compilare şi legare este prezentat în figura 3.4.2.1). Serverul de nume returnează clientului handle-ul şi identificatorul unic al serverului; în continuare clientul utilizează handle-ul ca adresă de destinaţie a mesajelor, în timp ce identificatorul unic este inclus în fiecare mesaj, şi folosit pentru a distinge un server dintre mai multe executate pe aceeaşi masina. Aceasta tehnica, a exportului şi importului interfeţei serverului oferă un grad mare de flexibilitate: serverul de nume poate asocia clienţii la servere în aşa fel încît să echilibreze încărcarea în sistem, poate contribui la mecanismul de autentificare al clienţilor, sau poate testa

BUPT

periodic serverele înregistrate, pentru a deînregistra automat serverele care nu mai raspund la apeluri. Tehnica are însă şi dezavantajul că necesită o circulaţie de mesaje în plus pentru exportul şi importul interfeţelor; dacă se menţine un singur server de nume pentru toti clienţii, într-un număr mare, serverul de nume poate fi sufocat de cereri, astfel încît sunt necesare mai multe servere de nume, a caror sincronizare pentru actualizarea informaţiilor creste şi mai mult regia de sistem.

Se prezintă în continuare, ca exemplu, modul de legare al clienţilor la servere adoptat în SUN RPC[Ste90] (RPC este o componentă în ONC= Open Network Computing). Mecanismul SUN RPC este asigurat de 3 elemente: (1) *rpcgen*, compilatorul care, primind descrierea interfeţei procedurilor generează codurile stub (server şi client); (2) *XDR (eXternal Date Representation)* standardul de codificare a informaţiei pentru compatibilitate între sisteme diferite; (3) o bibliotecă de funcţii run-time. Un serviciu, implementat de un server, este reprezentat de doi identificatori (numere întregi fără semn pe 32 biţi): numărul de program şi versiunea lui. Serviciile se înregistrează într-un superserver denumit *portmapper* cu urmatoarele informaţii: numarul de program, numărul versiunii, protocolul UDP sau TCP), numărul portului, numele serviciului şi numele rutinei de dispecerizare. Portmapper-ul este lansat ca un proces daemon la startarea sistemelor; printr-o comanda Unix, folosind diverse optiuni, se poate afişa lista serviciilor înregistrate, a serviciilor ce folosesc porturi UDP sau TCP, se pot deînregistra servicii. Procedurilor unui serviciu li se asignează numere. Numărul 0 se asignează unei proceduri speciale, care nu primeşte argumente şi nu calculează rezultate doar returnează controlul apelantului, generate automat de compilatorul rpcgen, şi folosită pentru a verifică dacă serviciul este in execuţie sau nu. Apelul unei proceduri la distanţă se face dupa crearea unui handle client (UDP sau TCP) care precizează adresa serverului (portul poate fi determinat de portmapper), numele de program, versiunea şi soclul clientului; dupa obţinerea handle-ului se pot apela proceduri, indicandu-se handle-ul clientului, numărul procedurii, argumentele şi rutinele xdr (filtre xdr) care serializează argumentele, respectiv deserializează rezultatele. Procesul de legare, descris în figura 3.4.3.1, decurge astfel: (1) la lansarea procesului server, stub-ul server creează un soclu, la care se asociază un port local; (2) se apelează funcţia de înregistrare a serviciului (svcÎregister) prin care se contactează portmapper-ul local pentru înregistrarea serverului; (3) se lansează clientul, care mai intai, contactează portmapper-ul de pe nodul indicat (clnt_creat()) pentru a determina portul serverului; (4) la apelul unei proceduri de la distanţa, stub-ul client serializează argumentele şi le transmite stub-ului server, folosind portul determinat la (3).

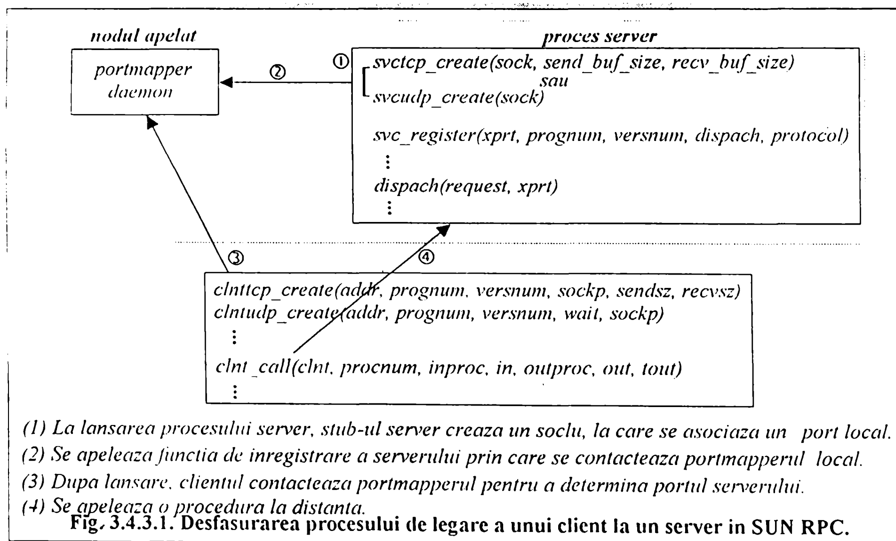


Fig. 3.4.3.1. Desfasurarea procesului de legare a unui client la un server in SUN RPC.

Procesul portmapper de pe nodul serverului este contactat de clienţi pentru a se localiza un program cu un anumit număr şi cu o anumita versiune. Clientul trebuie sa specifice în plus doar nodul serverului şi protocolul de transport. Procesul portmapper furnizează şi suport pentru tratarea cererilor tip broadcast catre un anumit program. O cerere broadcast poate primi mai multe raspunsuri, de la fiecare nod care conţine programul respectiv; raspunsurile conţin şi numerele porturilor asociate.

SUN RPC furnizează şi modalităţi de autentificare a clienţilor: formă implicita (fără autentificare), autentificare UNIX şi autentificare DES. Autentificarea Unix a clienţilor provoaca transmiterea în fiecare apel RPC a urmatoarelor campuri: o marca de timp, numele nodului, identificatorul de utilizator al clientului, identificatorul de grup efectiv al clientului şi lista identificatorilor tuturor grupurilor la care apartine clientul. Este sarcina serverului să examineze aceste cîmpuri pentru a determina dacă poate raspunde unui client sau nu.

BUPT

3.4.4. Probleme de concurenta

Mecanismul RPC este util în programarea distribuită, în medii eterogene chiar, dar nu incurajează calculul paralel, deoarece atunci cînd un client apelează o procedură de la distanța, el se blochează pe parcursul execuției procedurii. Aceasta poate cauza performanțe slabe dacă clienții și serverele nu utilizează mecanismul firelor; de exemplu un server poate apela, la rîndul lui, pentru implementarea unui serviciu, alte proceduri din alte servere și poate deservi alti clienți în timp ce așteaptă răspunsurile de la aceste servere. Folosirea firelor este o soluție adecvată în multe dintre problemele care apar datorită blocării într-un apel de procedură la distanță. Un server poate executa concurent mai multe apeluri ale clienților, fiecare apel într-un fir. Analog, un client poate crea mai multe fire, pentru a fi executate concurent; dacă unul dintre acestea se blochează într-un apel de procedură la distanța, celelalte îsi pot continua execuția.

3.4.5 Semantica RPC în prezența defectelor

Spre deosebire de procedurile locale, pentru care se știe întotdeauna că, la returnarea controlului, după apelul lor, au fost executate exact o data, în cazul procedurilor executate la distanță, dacă nu se returnează controlul dupa un anumit interval de timp este greu de precizat de cîte ori au fost executate. Există 5 situații care trebuie considerate în acest sens:
(1) Clientul nu poate localiza serverul (un exemplu este un client care încearcă să contacteze o versiune invechită a unui server, ce nu mai există); eroarea poate fi raportată prin generarea unei exceptii (cu dezavantajul că nu toate limbajele dispun de mecanismul exceptiilor).
(2) Mesajul de apel al procedurii se pierde. În acest caz, dacă nu se primește raspunsul (sau o confirmare) după un interval de timeout, se poate retransmite mesajul. Dacă și după un număr maxim de încercari nu se primește raspuns, se poate trage concluzia că serverul a cazut (și se poate adopta aceeași soluție ca la punctul 1).
(3) Mesajul de raspuns la procedura se pierde. În acest caz fie că se încearca o asemenea structură a cererilor incît sa fie idempotente, fie se asignează ficărei cereri un număr de secvență iar serverul reține rezultatele ultimelor cereri astfel încît la detectarea unei cereri duplicate, nu se mai execută procedura ci se transmite direct rezultatul. Un exemplu: în SUN RPC, dacă se folosește ca protocol de transport protocolul UDP, se retransmite automat o cerere dacă este necesar; timeout-ul se poate specifică de client (în funcțiile clntudp_create(). clnt_call()). După un număr maxim de încercari nereusite se returnează o eroare procesului client. Dacă se folosește ca protocol de transport protocolul TCP, se returnează o eroare clientului dacă conexiunea este terminată de server. Fiecărei cereri de execuție a unei proceduri a unui client i se asociază un identificator pe 32 biți care este trimis de client serverului; serverul returnează acest identificator clientului, astfel că răspunsul la o cerere nu se livrează decît după ce clientul verifică identificatorul primit. În plus, serverele UDP pot menține o zona cache a răspunsurilor trimise, astfel încît cererile duplicate nu generează noi execuții.
(4) Nodul serverului sufera o avarie grava. Deoarece în aceasta situație există posibilitatea ca (1) serverul sa fi suferit avaria înainte de a executa cererea sau (2) după execuție, dar înainte de a transmite răspunsul, se disting 3 semantici posibile: (1) *semantica cel-puțin-o-dată* care garantează că procedura a fost executată cel puțin o data, dar posibil de mai multe ori; este situatia ideală pentru cereri idempotente; pentru cereri neidempotente există mai multe variante, dintre care cea mai folosită este varianta *ultima-din-mai-multe*; (2) *semantica cel-mult-o-dată* care garantează că cererea cel mult o dată dar posibil nici o dată; (3) *semantica exact-o-dată* este este cerința ideala, dar este dificil de obținut.
(5) Nodul clientului sufera o avarie grava. În această situatie, dacă clientul a cerut execuția unei proceduri la distanță, și între timp a suferit o avarie, serverul poate ramane în execuție un timp nedefinit; astfel de procese active, fără ca rezultatele execuției lor sa fie așteptate de vreun proces se numesc *orfani*. Procesele orfane trebuie distruse deoarece, în afară de faptul că irosesc timp procesor, pot bloca accesul la resursele pe care le detin (semafoare, fișiere). Există patru tehnici de distrugere a orfanilor [Gos91][Tan89][Tan96]: (1) tehnica *exterminarii*, aceasta impune ca, la orice mesaj RPC, stub-ul client sa înregistreze într-un fișier jurnal (care trebuie sa supraviețuiască avariei) informații despre cerere. După reinitializarea nodului clientului, jurnalul este verificat și orfanii distrusi explicit. În afară de faptul că necesită resurse speciale (fișierul jurnal), tehnica poate sa nu funcționeze în unele cazuri particulare, precum partitionarea unei rețele (datorita unei parți care nu funcționeaza, de exemplu) cînd se poate localiza un server orfan, dar nu se poate distruge sau cînd un orfan a lansat în execuție alte apeluri RPC, aparînd astfel orfani nepoti, care nu mai pot fi detectați; (2) tehnica *expirării* este analogă, ca principiu, tehnicii time-out; la începerea execuției unei proceduri, serverul are alocat pentru execuție un timp maxim T. Dacă se depașește acest timp, stub-ul server cere stub-ului client o nouă cuantă T; în cazul în care clientul nu mai există, serverul se distruge. Tot ceea ce trebuie sa faca un client care se reinitializează, este să aștepte un interval de timp T; astfel este sigur că orfanii sunt distruși; (3) tehnica *reincarcării* divizează timpul în perioade, numite epoci; cînd un client este reinitializat, el trimite (prin broadcast) catre toate nodurile un mesaj de start al unei noi epoci. La primirea unui astfel de mesaj, toti orfanii

BUPT

sunt distruşi. Evident, şi aceasta tehnică nu detectează toti orfanii dacă reţeaua este partitionată în urma unui defect hardware; totuşi cînd vor putea returna un răspuns vor raporta o epoca diferita de cea curenta; (4) tehnica *reincarcării lente* este o varianta mai puţin restrictiva a tehnicii reincarcarii. Astfel, cînd un nod primeşte un mesaj de început al unei noi epoci verifică dacă are servere în execuţie şi, în caz afirmativ, clienţii lor. Serverele se distrug numai dacă clienţii lor nu pot fi localizati.

3.4.6. Protocoale pentru RPC

Semantica unui apel de procedură la distanţă nu trebuie sa fie legată de implementarea nici unui protocol; rutinele RPC care se apelează la nivelul stub-urilor trebuie sa fie disponibile indiferent de clasa protocolului de transport utilizat. Astfel, o primă clasă de protocoale pe care se pot construi apeluri ale procedurilor la distanţă sunt protocoalele de nivel transport care asigură un transfer de date în flux, sigur, de exemplu TCP. Protocolul TCP este utilizat în SUN RPC, DCE RPC, Microsoft RPC. Protocoalele bazate pe conexiuni, cum este TCP, prezintă avantajul că implementarea RPC nu trebuie să trateze defectele, nu trebuie să utilizeze confirmări; protocolul orientat pe conexiune se ocupă de aceasta. Cînd aplicaţia lucrează într-o reţea de mare intindere geografică (deci nu o reţea locală) aceasta este soluţia cea mai bună. Cealaltă clasă de protocoale care pot fi utilizate de implementările RPC cuprinde protocoalele nesigure, la nivel de datagrame, de exemplu UDP; şi protocolul UDP poate fi utilizat în SUN RPC, DCE RPC, Microsoft RPC. Utilizarea unui protocol cunoscut cum este UDP (de fapt IP, care sta la baza lui) oferă avantajul unui protocol deja elaborat, disponibil pe mai multe arhitecturi, folosit aproape în totalitate de sistemele Unix, care asigură comunicaţia directă între multe reţele deja existente dar şi dezavantajul că protocolul IP, nefiind proiectat ca un protocol end-user nu oferă o eficienţă ridicată. Alternativa este de a folosi un protocol specializat, care sa fie proiectat avînd ca scop principal utilizarea lui în reţele locale. Birell şi Nelson ([BN84]) au propus un protocol, adoptat apoi de sistemul Sprite ([OCD88]), avînd drept scop implementarea protocolului în comunicaţii prin RPC. În cadrul acestui protocol, apelurile de proceduri sunt impartite în două grupuri: *simple* şi *complicate*. Cele simple se disting de cele complicate prin faptul că argumentele lor, ca şi rezultatele incap într-un singur pachet şi prin frecvenţa lor ridicată. Un pachet trimis printr-un apel de procedură simplă trebuie sa cuprinda un identificator al apelului, numărul procedurii apelate şi argumentele; un pachet de răspuns cuprinde acelaşi identificator din apel şi rezultatele. Pentru apeluri simple, pachetele cu rezultate sunt folosite drept confirmări ale pachetelor cu cereri. Există trei situaţii cînd se retransmite o cerere simplă, în urma detectării unui time-out (cînd nu s-a primit răspuns după un interval maxim): (1) cererea s-a pierdut; (2) nu s-a pierdut nici cererea, nici răspunsul, dar execuţia durează mai mult; în aceasta situaţie se transmite o confirmare explicita de catre nivelul de transport; (3) s-a pierdut răspunsul; procedură nu se mai execută inca o data, se trimite doar răspunsul. Apeluri complicate se utilizează în 2 cazuri: (1) execuţia procedurii durează mult sau apelurile sunt rare. În acest caz există două strategii pentru retransmisii şi confirmări: (a) emiţătorul unui pachet este responsabil pentru retransmisia unui pachet pînă la primirea confirmării; după primirea confirmării, aşteaptă rezultatul, eventual trimitînd cate un pachet de probă, pentru a vedea că receptorul mai există; (b) receptorul transmite automat o confirmare dacă nu poate genera un pachet în intervalul de timp maxim pentru o r

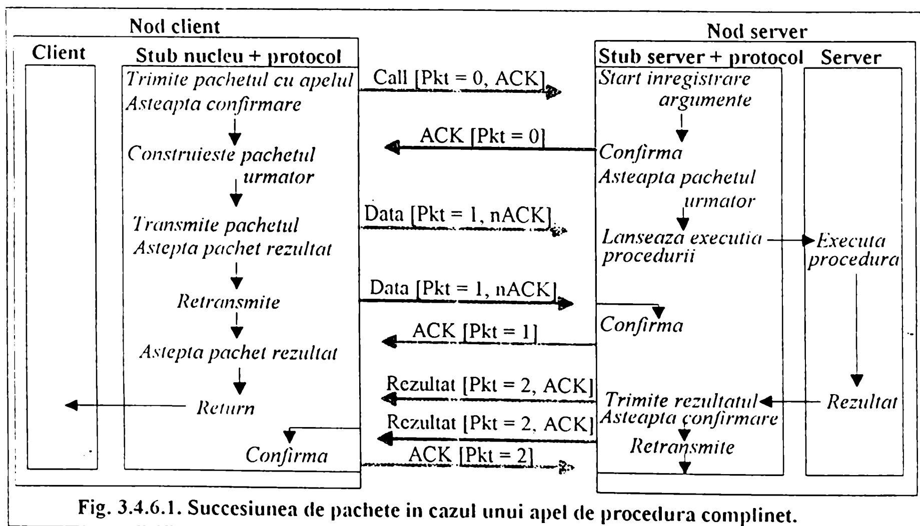


Fig. 3.4.6.1. Succesiunea de pachete in cazul unui apel de procedura complinet.

BUPT

etransmisie; (2) al doilea caz cînd se considera un apel complicat este cel în care argumentele sau rezultatele unei proceduri nu incap într-un singur pachet. În acest caz se confirmă numai ultimul pachet din seria rgumentelor sau răspunsurilor iar pachetele dintr-o serie se numeroteaza; în acest mod, în cazul pachetelor pierdute se evita retransmisia întregii serii de pachete.

În figura 3.4.6.1 se arată succesiunea de pachete interschimbate în cadrul unei proceduri la distanţă considerată complicată deoarece execuţia ei durează un interval de timp mai mare decît cel stabilit ca maxim pentru a initia o retransmisie iar argumentele şi rezultatele nu incap într-un singur pachet.

3.4.7. Extensii ale mecanismului de bază RPC

Mecanismul RPC de bază a fost extins în Sprite Network Operating Systems [OCD88][Gos91] pentru a permite comunicaţii între mai multi clienţi şi mai multe servere; în Sprite, mecanismul RPC este implementat la nivel de nucleu catre nucleu într-un mod asemanator cu ceea ce s-a descris mai înainte. Este posibil însă, ca un nucleu client sa ofere servicii la mai mult de un proces care a apelat o procedură la distanţă şi că un server sa deservească mai multi clienţi în acelaşi timp. Ambele extensii pot fi implementate pe baza notiunii de *canale* care conectează procesele client la procesele server; aceste canale, rezervate în număr fix pentru fiecare calculator, sunt multiplexate între clienţi (fig.3.4.7.1). Un calculator server are mai multe procese server; procesele server păstrează informaţii despre canalele utilizate de procesele client şi nu despre procesele client. Un proces client poate utiliza un canal atunci cînd el este liber. Nucleul cu rolul de server este cel care asignează un proces server la un canal al unui client, pentru a răspunde la o serie de apeluri de proceduri; selectia procesului server care trebuie sa răspunda la o cerere a unui client se face pe baza identificatorului procesului server care a fost utilizat pentru a tratā ultima cerere.

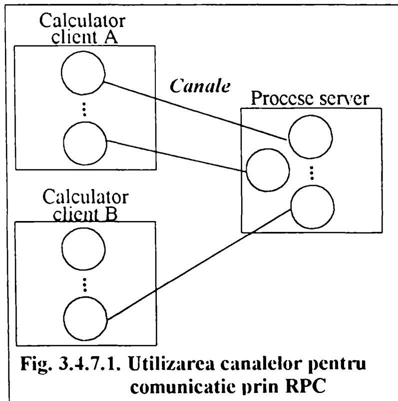


Fig. 3.4.7.1. Utilizarea canalelor pentru comunicatie prin RPC

O altă extensie care a fost adusă mecanismului RPC în sistemele de operare distribuite experimentale este posibilitatea de a difuza o cerere RPC la mai multe servere, pentru a obţine un grad sporit de siguranta în sistemele în care nodurile se pot defecta independent. Astfel, în sistemul de operare HPC University of Rochester [BF85] s-a introdus conceptul de apeluri de *proceduri replicate*, alaturi de acela de *trupa*. O *trupa* este un set de module replicate; -trupele se pot executa pe mai multe calculatoare. Cînd o trupa, reprezentînd un modul client, cere un serviciu, fiecare membru al trupei server execută procedura necesară şi fiecare membru al trupei client primeşte un răspuns. Un program construit pe baza apelurilor de proceduri replicate va continua sa lucreze în prezenţa defectelor atîta timp cît există cel puţin o replică (copie) al fiecărui modul. Un ringmaster se foloseşte pentru a lega (bind) nume la trupe.

Fig. 3.4.7.2 arată două trupe şi un ringmaster. Fiecare trupa este implementată ca un obiect HPC cu o interfaţa multicast. În interiorul trupei, membrii ei sunt interconectăţi la capatul interfeţei multicast. Cele două trupe sunt conectate în exterior prin intermediul extensiei interfeţei multicast; fiecare trupa are o interfaţă similara şi pentru comunicaţia cu ringmasterul. Acesta are o interfaţă multiplexată pentru a comunică cu un număr variabil de trupe; fiecare componentă a interfeţei este o extensie multicast. Toate trupele au interfeţe externe analoage, indiferent de cati membrii replicati au. Complexitatea membrilor replicati ai unei trupe se izolează la interiorul trupei (guvernată de un shell), în timp ce complexitatea datorată unui număr variabil de trupe şi interacţiunilor lor se izolează la nivelul imediat superior. Astfel, în HPC se permite trupelor să fie construite independent de utilizarea lor şi să fie utilizate independent de constructia lor.

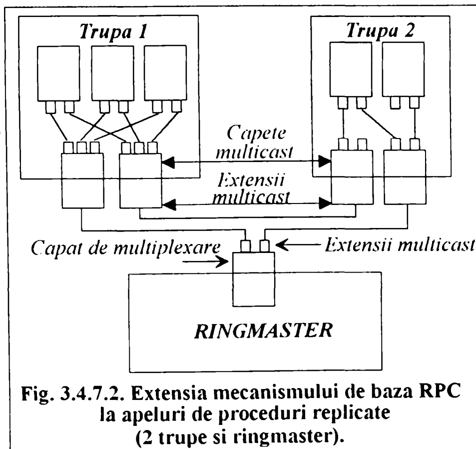


Fig. 3.4.7.2. Extensia mecanismului de baza RPC la apeluri de proceduri replicate (2 trupe si ringmaster).

BUPT

3.4.8 Aspecte critice în proiectarea comunicării prin RPC.

Tannenbaum a identificat ([Tan93][Tan96]) în desfăşurarea unui apel de procedură la distanţă etapele prezentate în fig. 3.4.8.1. Aceste etape formează *calea critica*. Unele aspecte ale acestor etape, considerate critice, pot avea o influenţă considerabila asupra timpului total de execuţie al unei proceduri la distanţă.

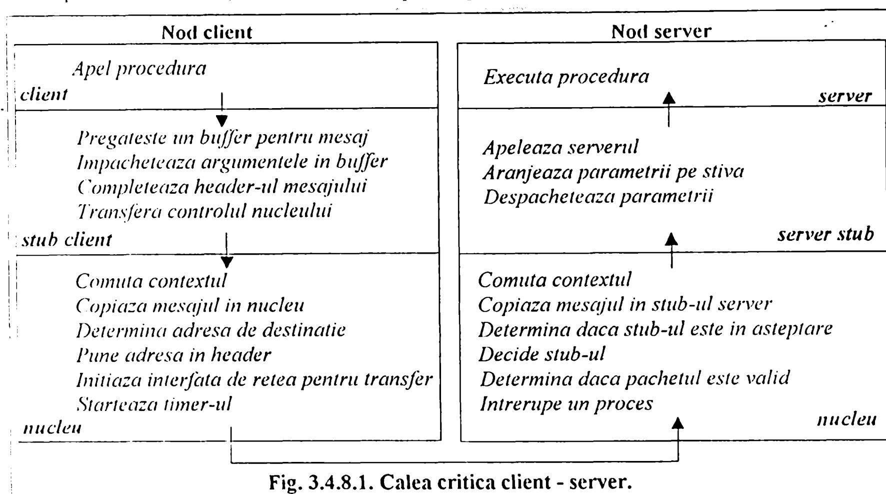


Fig. 3.4.8.1. Calea critica client - server.

Astfel, problema copierii continutului mesajelor de transmis are o importanţă majoră deoarece în funcţie de structurarea software-ului şi de facilităţile hardware pot fi necesare de la 2 pînă la 8 faze de copiere a mesajelor; în acest sens o importanţă mare o prezintă urmatoarele aspecte: (1) posibilitatea ca interfaţa de reţea sa acceseze prin DMA zona de memorie din stub-ul client care conţine mesajul de transmis; (2) prezenţa facilităţii *scatte/gather* la interfaţa de reţea (facilitatea *scatter/gather* a fost definita în sectiunea 2.3.); aceasta este utilă deoarece sistemul de operare poate construi numai header-ul pachetelor, lasînd mesajul propriu zis în stub-ul client; (3) dacă se utilizează memorie virtuală, se poate include pagina de memorie care conţine un mesaj receptionat în spatiul de memorie al stub-ului server. Alte aspecte care pot aduce o îmbunătăţire a performantelor mecanismului RPC este folosirea algoritmilor de analiza (*sweep timers,* vezi sectiunea 2.3.) şi reducerea regiei de sistem datorată gestiunii proceselor în nodul serverului.

Pentru a se exemplifica directiile în care se încearcă optimizarea mecanismelor RPC, se prezintă în continuare modalităţile de optimizare ale mecanismului RPC Peregrine (realizat la Rice University, Houston, Texas, [JW93]), ale cărui performanţe sunt remarcabile (timpul de trecere pentru o procedură nula este de 573us faţă de 1100us la Amoeba şi 1097us la Cedar) şi apoi se va prezenta o comparaţie între cîteva implementari RPC alături de performanţele lor (publicate).

Peregrine RPC a fost proiectat în vederea obţinerii unor performanţe superioare în ceea ce priveste comunicaţia interprocese, mentinînd facilităţi legate de comunicaţia prin proceduri precum argumente şi rezultate de tipuri arbitrare. Implementarea (şi masuratorile) a fost facuta pe baza unor staţii Sun 3/60 conectate într-o reţea Ethernet, prin controlere de reţea AMD Am/7990 LANCE. Protocolul RPC se situează imediat deasupra protocolului IP şi este asemănător protocolului Cedar RPC, cu excepţia faptului că utilizează un protocol blast pentru mesajele multipachet. În Peregrine, este responsabilitatea clientului sa convertească argumentele la reprezentarea serverului şi rezultatele la reprezentarea sa proprie, eliberînd astfel serverul de aceasta sarcină, care poate conduce în unele situaţii la degradarea performanţelor serverului.

În Peregrine RPC se realizează creşterea performanţelor prin urmărirea obiectivelor de mai jos: (1) Argumentele şi rezultatele se transmit direct din spatiul de adrese utilizator (client sau server) evitînd astfel copierile intermediare. Se utilizează facilitatea gather DMA a controlerului de reţea (controler Ethernet): primind o lista de segmente, fiecare segment fiind specificat prin adresa de început şi lungimea, controlerul poate transmite un singur pachet ale cărui date constau în combinarea segmentelor de mai sus. (2) Nu se realizează conversia datelor (argumente şi rezultate) cînd clientul şi serverul utilizează aceeaşi reprezentare (spre deosebire, de exemplu de SUN RPC care utilizează intotdeauna reprezentarea standard). (3) Pachetele se trimit utilizînd headere deja alocate şi completate evitînd astfel aceste operaţii la fiecare apel. (4) Nu se salvează starea firelor între apeluri în server; deci nu este nevoie de salvarea şi restaurarea registrelor între apeluri în cadrul serverului şi nici de salvarea stivei firului. (5) Argumentele sunt mapate în spatiul de adrese al serverului, deci nu sunt copiate. Se utilizează facilitatea de a remapa paginile într-un spatiu de adresa numai prin modificarea intrarilor corespondente din tabela de pagini, procedură suficientă pentru Sun 3/60 (alte

BUPT

sisteme mai necesită şi modificarea intrarilor corespunzatoare din TLB= translation lookaside buffer). (6) Argumentele apelurilor multipachet sunt transmise încît nu este necesară copierea lor în calea critica; ele sunt fie copiate în paralel cu transmisia în reţea, fie sunt mapate direct în spatiul serverului.

În figura 3.4.8.2 se prezintă o comparaţie între modalităţile utilizate pentru optimizarea performanţelor de cateva sisteme RPC şi sisteme bazate pe comunicaţia prin mesaje dar care construiesc apeluri sistem asemanatoare apelurilor de proceduri. În cea de-a două categorie pot fi considerate comunicaţiile din Amoeba şi din V Kernel.

Pentru compararea performanţelor sistemelor RPC se folosesc: timpul de trecere al unui apel de procedură nula şi productivitatea obţinuta în urma unei serii de apeluri de proceduri cu argumente mari.

Sistem	Caracteristici ale implementarii	Daca se foloseşte facilitatea gather DMA si header template	Cum se face conversia datelor	Cum se transmit argumentele în spatiul de adrese al serverului	Cum se realizeaza execuţia procedurilor la nivelul serverului
V System	Nucleul se bazeaza pe comunicaţii prin mesaje	In unele implementari; se foloseşte un header template		Prin copiere; pentru argumente multipachet copierea se face în paralel cu transmisia	
Sprite	RPC folosit pentru comunicaţii nucleu catre nucleu	Intr-un mod limitat; se foloseşte un header template	la nivelul receptorului	Prin copiere; pentru argumente multipachet copierea se face în paralel cu transmisia	Un set de fire daemon ale nucleului trateaza cererile; stiva şi registrele firelor se salveaza şi restaureaza
X-Kernel	Implementarea se compune dintr-un numar de protocoale de nivel jos	Fiecare protocol foloseşte un header template	Datele se transmit în formatul emiţătorului, cu un bit indicind arhitectura lui	Argumentele mici prin copiere; pentru cele mari se evita copierea	Procedura se executa de un fir server se-lectat dintr-un set de fire prealocate; firele persista în urma exe-cutiei; registrele şi stiva se salveaza şi restaureaza
Firefly	Implementarea s-a facut pe un sistem multiprocesor	Se foloseşte tran-smisia prin DMA direct din bufferul utilizatorului; se foloseşte un header template construit în faza de legare	Se negociaza reprezentarea datelor în faze de legare	Se evita copierea	Un set de fire trateaza cererile
Amoeba	Nucleul se ba-zeaza pe comu-nicaţii prin me-saje; RPC este constituit pe ba-za unor primitive sincrone prin stub-uri care se pot genera şi automat	In nucleu; Amoeba reconstruieste complet header-ul pachetelor fără a utiliza un template la fiecare transmisie	Conversiile sint lasate în seama stub ului	Prin copiere	Se folosesc stub-uri
Sun RPC	Implementarea s-a facut pe baza unor funcţii de biblioteca scrise deasupra nucleului Unix	Nu este cazul	Se foloseşte transmisia într-o reprezentare standard a datelor	Prin copiere	Se folosesc stub-uri

Fig. 3.4.8.2. Comparaţie între citeva sisteme bazate pe RPC şi pe comunicaţii prin mesaje.

BUPT

3.4.9. Exemple

A. Modelul RPC în DCE

Transmiterea parametrilor şi reprezentarea datelor

În DCE, RPC este un mecanism de bază; implementarea DCE RPC foloseşte un limbaj specific pentru definiţia serverelor, Interface Definition Language (IDL) care este de fapt un supraset al partii declarative din limbajul C (corespunzînd portiunii de fişier header al unui program C). Functiile run-time RPC sunt oferite tot în limbajul C (similar altor componente DCE). IDL oferă pentru specificarea tipurilor parametrilor procedurilor facilităţile limbajului C. Diferentele între reprezentarea datelor la client şi server se rezolva prin tehnica *receiver makes right*; aceasta implică transmiterea datelor în reprezentarea emiţătorului şi adaptarea lor la reprezentarea receptorului la nodul destinatar.

Legarea clienţilor la servere

Desfasurarea unui apel de procedură la distanţă în DCE este reprezentată în figura 3.4.9.1. La initializarea unui server se determina ce protocol se utilizează (TCP/IP sau UDP/IP), se înregistrează interfaţa procedurilor oferite la un manager local de RPC-uri, se exporta informaţii despre server la CDS, şi, în final, se aşteaptă sosirea unor cereri. Pentru a apela o procedură la distanţă, clientul foloseşte o sintaxa normala, ca pentru un apel local. Totusi, datorita stub-ului client generat din IDL, se execută o rutina interna; aceasta contactează serviciul de directoare CDS pentru a localiza un server adecvat (informaţiile de intrare sunt numele logic al serverului şi interfaţa dorita, iar informaţia de ieşire este adresa serverului). Apoi se codifică parametrii de intrare şi se transmit serverului.

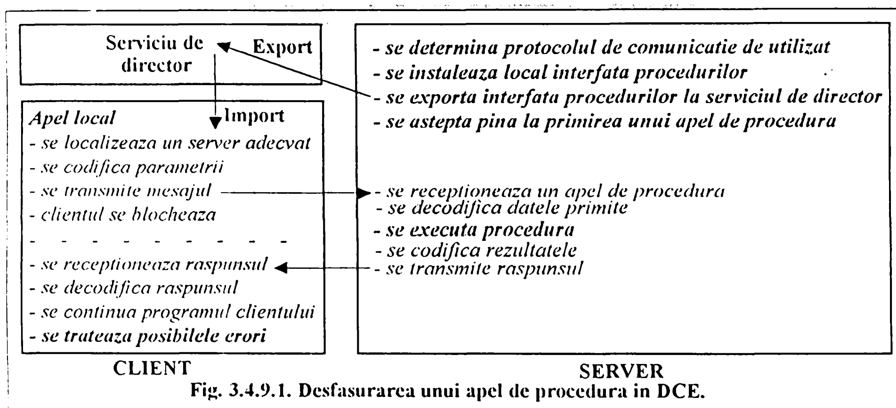


Fig. 3.4.9.1. Desfasurarea unui apel de procedura in DCE.

Concurenta

Existînd fire, este posibil sa se implementeze servere multifir. Aceasta implică doar setarea unui parametru adecvat la initializarea serverului; apoi un ansamblu de servere concurente este alocat (static). Programatorul este cel care trebuie sa tină seama, însă, de corecta sincronizare a firelor, în cazul modificării datelor partajate. În ceea ce priveste clientul, firele trebuie startate explicit, pentru a realiza apeluri asincrone la mai multe servere.

Semantica RPC

Aplicaţiile pot alege între diferite tipuri de semantica. Astfel, semantica implicită, cel-mult-o-dată asigură execuţia unui apel chiar dacă mesajele sunt pierdute temporar (printr-o tehnică de retransmisie combinată cu detectia mesajelor duplicate). Desi caderea nodurilor nu este tolerată, pierderea mesajelor poate fi mascată în acest mod. Alte semantici care pot fi selectate furnizează garantii mai puţine în prezenţa defectelor, dar realizează o crestere a eficienţei transmisiei.

Securitatea

Siguranţă comunicaţiilor RPC se asigură prin serviciul de securitate. Intai, clientul şi sereverul execută un protocol de autentificare în cooperare cu serverul pentru securitate. În aceasta faza, ambii isi validează reciproc identitatea pe baza unui mecanism de criptare cu cheie privata. În faza a doua, cînd trebuie executata o procedura, serverul verifică autorizarea unui client pe baza unei liste de control al accesului locală. În plus, datele pot fi optional criptate, pentru a asigură o protecţie completa şi în cursul comunicaţiilor.

BUPT

Dezvoltarea aplicaţiilor distribuite cu DCE RPC

Dezvoltarea aplicaţiilor distribuite cu DCE RPC necesita trei etape [BGH93]: definirea interfeţei IDL, implementarea serverului şi implementarea clientului. În continuare, prezentarea acestor etape se va face pe baza unui mic exemplu care constă într-un client ce adresează cereri unui server de gestiune a produselor într-o întreprindere.
Prima etapa consta în definirea unei interfeţe IDL care trebuie sa specifice toate procedurile serverului. Interfaţa constă într-un header cu un număr de interfaţă unic (generat automat) şi cu informaţii despre versiune şi într-un corp care cuprinde definiţia funcţiilor ca în limbajul C, cu specificarea tipurilor parametrilor (de intrare şi/sau de ieşire). Pentru exemplul luat, definiţia interfeţei este cea din figura 3.4.9.2.
A doua etapă este implementarea serverului (fig.3.4.9.3). Faza de initalizare a serverului este implementată conform celor expuse anterior.
Implementarea clientului este independentă de aspectele de distribuţie (fig.3.4.9.4).

```
[
uuid(876c2a20-100a-145c-1678-040083d67921)
version (1.0)
]
interface DateProductie {// interfata pentru date productie
  import "globaldef.idl";// import al definitiilor generale
  const long ProdMax=30;// numar maxim de produse
  typedef [string] char *string;// tipul string
  typedef struct {
     string NumeProdus;// nume produs
     string DetaliiProdus;// informatii despre produs
     Plan PlanDeProductie; // tip definit in globaldef.idl
  } DescriereProdus;// tip de date pentru descriere produs
long CerereProdus (// procedura ce poate fi apelata de la distanta
  [in] string NumeProdus[ProdMax], // parametru intrare
 [in] DescriereProdus *pd[ProdMax], // parametru de iesire
 [out] long *stare); // parametru de iesire
}
```

Fig.3.4.9.2 Definirea interfetei

```
#include "dateproductie.h"// generat de compilatorul IDL
#define DirServer "/.:/ServerProductie"// numele directorului serverului
#define MaxApelConc 5// numar maxim de apeluri concurente
main() {
unsigned stare;// stare returnata
rpc_binding_vector_t *bVec;// vector de handle-re
//*** Alte initilizari ***
//*** Preia vectorul handle ***
rpc_server_inq_bindings(&bVec,&stare);

//*** Se inregistreaza local interfata ***

 rpc_ep_register (DateProductie_v1_0_s_ifspec, bVec,NULL,NULL,&stare);

//*** Se exporta interfata la CDS ***
rpc_ns_binding_export (rpc_c_ns_syntax_default, DirServer,
  DateProductie_v1_0_s_ifspec, bVec,
  NULL, &stare);
 //*** Accepta cereri concurente ***
rpc_server_listen (MaxApelConc, &stare);
}
```

Fig.3.4.93 Serverul

BUPT

```
#include "dateproductie.h"
main() {
string produse[ProdMax];            // nume produse
DescriereProdus *dp[ProdMax];       // descrieri cerute
long stare,rq;
PreiaDate (produse);                // functie specifica aplicatiei
rq=CerereProdus (produse,dp,&stare);// RPC
// ... Se verifica starea si se trateaza erorile
```

Fig. 3.4.9.4 Clientul

B. Microsoft RPC

Microsoft RPC se bazează pe DCE RPC; difera totusi de aceasta prin urmatoarele:
(1) În biblioteca de funcţii runtime, denumirile sunt modificate în concordanta cu conventiile Microsoft; de exemplu funcţia rpc_string_binding compose() devine RPCStringBindingCompose();
(2) funcţiile raportează informaţii de stare prin valoarea lor de retur; în OSF se foloseşte o variabila de ieşire;
(3) Tipurile de date primesc nume, utilizandu-se typedef pentru tipurile structurate;
(4) Unele funcţii primesc argumente în plus, specifice sistemului NT, precum descriptori de securitate;
(5) Sunt adăugate funcţii şi macro-uri noi pentru tratarea exceptiilor.

Microsoft RPC cuprinde componentele traditionale mecanismelor RPC: (1) compilatorul IDL (versiunea Microsoft se numeste MIDL). Compilatorul primeşte ca intrare două fişiere (.IDL şi optional .ACF) pentru a genera codurile stub pentru server şi client; acestea sunt apoi compilate şi linkeditate cu aplicaţiile client şi server (fig. 3.4.9.5); (2) bibliotecile de funcţii runtime, utilizate de aplicaţiile server pentru a-şi anunta serviciile şi de aplicaţiile client, pentru a localiza şi contacta serverele. Bibliotecile Microsoft necesare, per total, sunt: rpcr4.lib (furnizează funcţiile runtime), rpcns4.lib (furnizează funcţiile pentru serviciile de nume), libcmt.lib (furnizează funcţiile C standard) şi Kernel32.lib (furnizează funcţiile pentru fire).

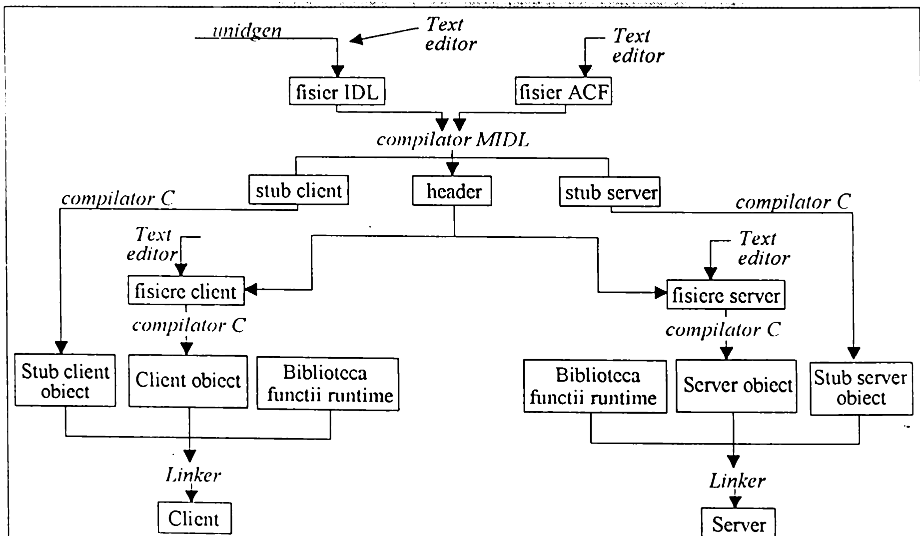


Fig. 3.4.9.5. Dezvoltarea aplicatiilor client-server pe baza mecanismului RPC in Microsoft RPC.

BUPT

Legarea clienților de servere

Fig. 3.4.9.6 arată informațiile cheie în procesul de legare al clienților la servere; acesta include urmatoarele operații (fig. 3.4.9.7):

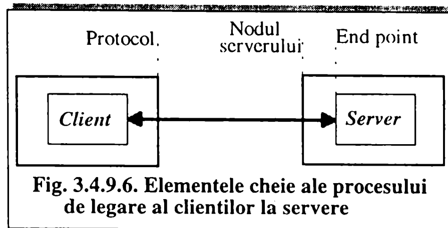


Fig. 3.4.9.6. Elementele cheie ale procesului de legare al clientilor la servere

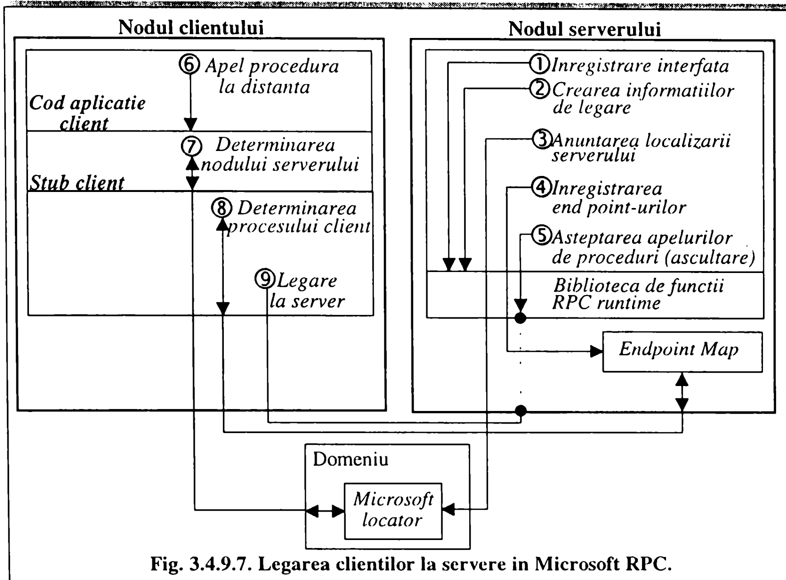


Fig. 3.4.9.7. Legarea clientilor la servere in Microsoft RPC.

A. **La initializarea unui server:**

(1) Înregistrarea interfeței serverului, în scopul realizării legaturii dintre handle-ul interfeței, produs de compilatorul MIDL și identificatorul sau unic UUID (UUID este o prescurtare de la Universal Unique Identifier) de catre rutinele RPC runtime; mai tîrziu clienții vor putea determina dacă sunt compatibili cu serverul.

(2) Crearea unor informații pentru legare; pentru aceasta trebuie stabilit unul sau mai multe protocoale de comunicație și eventual end-point-ul la care va asculta serverul sosirea apelurilor de proceduri. Pentru fiecare protocol posibil, end-point-ul are un format specific. Astfel, pentru *ncacn_np* (Named Pipes), end-point-ul este un nume de canal în formatul \PIPE\<nume>canal>, pentru *ncacn_cp_tcp* (conexiune PCP/IP) este portul TCP (string) iar pentru *ncacn_spx* (conexiune SPX) este un soclu cu valoare între 1 și 65536 (string). Deoarece Windows 95 nu are implementată partea de server pentru Named pipes, protocolul ncacn_np nu poate fi folosit în rețea în Windows 95.

(3) Plasarea informațiilor pentru legare într-o baza de date gestionată printr-un serviciu denumit Name Service, pentru ca clienții sa aiba acces la ele (Microsoft denumeste serviciul de nume *locator).* Biblioteca de funcții runtime conține un set de funcții generale (Name Service Independent Routines), care comunică cu locatorul pentru a plasa informații în baza de date; datorita nivelului de abstractizare al acestor funcții (cel mult independente de un serviciu de nume particular), Microsoft RPC poate fi configurat astfel încît sa utilizeze, de exemplu, cînd comunică cu sisteme DCE, serviciul DCE Call Directory Service. Microsoft RPC nu impune aplicațiilor distribuite folosirea serviciului de nume, desi pentru aplicațiile complexe, aceasta varianta este recomandata. Există și alte două modalități de a lega clienți la servere: (a) legarea directă, clientul cunoaste localizarea serverului, în codul clientului și serverului; (b) utilizarea unei metode specifice aplicației prin care serverul isi anunta identitatea clienților.

BUPT

(4) Plasarea informaţiilor despre end-point-uri într-o bază de date specială a nodului serverului, numita *harta end-point-ului* (End-Point Map); un server special, End-Point Mapper este responsabil pentru tratarea unei cereri RPC cu un handler de legare partial determinat. End-Point Mapper-ul încearca sa localizeze un server local, care sa ofere o interfaţă identificată prin acelaşi UUID; dacă reuseste acest lucru, atunci completează structura de date asociată handle-ului de legare. Din acest moment, handle-ul fiind construit, urmatoarele cereri nu mai contactează End Mapper-ul.
B. **La apelul unei proceduri de catre un client** (Etapele 6-9 corespund operaţiilor executate de client pentru legarea la server)
(6) Clientul execută un apel de procedură la distanţă;
(7) Prin intermediul locator-ului, stub-ul client determina nodul serverului;
(8) Prin intermediul End-point Mapper-ului, clientul determina identitatea serverului.

Dezvoltarea aplicaţiilor cu Microsoft RPC

Dezvoltarea aplicaţiilor cu Microsoft RPC se face urmărind aceleasi etape prezentate în sectiunea anterioara. Aceste etape pot fi ilustrate cu un exemplu simplu. În fig. 3.4.9.8 se arată configuratia fişierului. IDL al exemplului prezentat în fig. 3.4.9.9 prezinta codul funcţiei main a serverului. Codul clientului este independent de aspectele de distributie.

```
/* fisierul exemplu.idl */
[
uuid (47883460-bafb-11cf-a9do-0000c012d29d),  /* atribut uuid */
version(1.0),
pointer_default (ref)                    /* tratarea implicita pentru pointeri este p
]
interface exemplu                        /* interfata pentru exemplu */
{
    const unsigned short size=20,        /* dimensiunea maxima a vectorilor */
    typedef long array[size],            /* tip de date */
    void sum (
        [in]  array a1,      /* pointer intrare */
        [in]  array a2,      /* parametru intrare */
        [out] array a        /* parametru iesire */
    );
}
```

Fig. 3.4.9.8 Continutul fisierului exemplu.idl

```
/* fisierul server.c */
#include <stdio.h>
#include "exemplu.h"                    /* generat de compilatorul MIDL */
main() {
  unsigned long stare;                  /* cod de eroare */
  rpc_binding_vector_t *bvec            /* set de handle pentru legare */
  unsigned char * name;,                /* nume pentru serviciu */
/* inregistrarea interfetei */
stare=RpcServerRegisterIf (exemplu_v1_o_s_ifspec, NULL, NULL);
/*** cod pentru verificare stare ***/
/* crearea informatiilor de legare */
stare=RpcServerUseAllProtseqs (RPC_C_PROTSEQ_MAX_REQS_DEFAULT, NULL);
/*** cod pentru verificare stare ***/
/* obtinerea vectorului de handle-re; se incheie astfel etapa 2 din fig. 3.4.9.7 */
stare=RpcServerInqBindings (&bvec);
/** cod pentru verificare stare **/
/* se face cunoscut in baza de date nodul serverului */
name= (unsigned char *)getenv("EXAMPLE SERVER ENTRY");
STARE=RpcNsBindingExport(RPC_C_NS_SYNTAX_DEFAULT, name, exemplu_v1_o_s_ifspec, bvec
NULL);
/* cod pentru verificare stare */
```

Fig. 3.4.9.9. Codul functiei main() a serverului

BUPT

3.5. Comunicații de grup

O caracteristica a modelului RPC prezentat în sectiunea 3.4 este aceea ca implică în comunicare două parti: clientul şi serverul. Există însă aplicații în care este necesară implicarea mai multor procese, pentru care modelul RPC nu mai este adecvat. Ca exemplu de astfel de aplicații pot fi enumerate: serverele replicate de fişiere care asigură un serviciu de gestiune fişiere tolerant la defecte, sistemul de marketing electronic, protocoale two-phase, prelucrări distribuite, aplicații tip conferinta. În aceasta sectiune vor fi prezentate mecanisme de comunicații de grup, în cadrul carora un mesaj poate fi trimis mai multor receptori prin aceeaşi operație.

3.5.1. Probleme specifice de comunicații de grup

Un *grup multicast* este un set de procese care reprezintă destinatia aceluiasi mesaj ([GS91]). Mesajele pot fi generate de catre una sau mai multe surse, iar procesele destinație pot fi rezidente pe unul sau mai multe noduri, nu neaparat distincte. *Comunicatiile de grup (sau multicast)* se refera la cominicații de tipul unul-la-mai-multi, în contrast cu comunicația punct-la-punct prezentată pînă acum. Grupurile sunt entități dinamice; pot fi create şi distruse grupuri, iar componența grupurilor poate fi modificata. Un proces poate fi membru la mai multe grupuri în acelaşi timp. În cadrul unui grup, procesele pot fi egale sau poate fi creată o ierarhie între procese; de exemplu, un proces poate indeplini rolul de coordonator, în timp ce celelalte pot indeplini aceeaşi sarcina de lucru. Primul model prezinta o simetrie; dacă unul dintre procese dispare, grupul devine mai mic, dar continua sa lucreze. În cel de-al doilea, disparitia coordonatorului pune în pericol întregul grup. În schimb, în primul model, luarea unei decizii este o operație complicată, care implică un proces de votare.

Pentru *gestiunea grupurilor* (crearea, stergerea grupurilor, modificarea componenței grupurilor) există două soluții:

(1) o soluție *centralizata*, în cadrul căreia un server de grupuri primeşte şi tratează toate cererile care se refera la componența grupurilor; serverul de grupuri trebuie sa mentina o bază de date cu grupurile create şi componența lor. Soluția, desi simpla şi usor de implementat, prezinta dezavantajul că, în cazul disparitiei serverului de grupuri, gestiunea grupurilor este compromisa, iar grupurile trebuie reconstruite.

(2) o soluție *distribuita*, în cadrul căreia un proces transmite un mesaj tuturor proceselor din toate grupurile la intrarea sa într-un grup, ca şi la parasirea unui grup. Grupurile pot fi *inchise*, caz în care numai membrii unui grup pot transmite mesaje grupurilor sau *deschise*, caz în care orice proces din sistem poate transmite mesaje la orice grup. Grupurile inchise sunt utile în aplicații de prelucrare paralela (de exemplu, procesele care implementează un joc de sah pot construi un grup inchis: au acelaşi scop, desi nu interacționează cu exteriorul), iar grupurile deschise sunt caracteristice serverelor replicate, cînd un proces care nu este membru al grupului serverului (un client) trebuie sa poată sa le trimita un mesaj.

O problema specifică comunicațiilor de grup este *adresarea grupurilor;* dacă sistemul hardware (rețeaua) suportă adrese multicast, adresa fiecărui grup poate fi asociată unei adrese multicast. Dacă nu există această posibilitate, atunci se pot transmite mesajele prin broadcast, ramînînd ca fiecare nucleu al unui nod să ignore mesajul, dacă nici un proces din acel nod nu face parte din grupul destinatie. Dacă nu este disponibilă nici transmisia broadcast, atunci nucleul emițătorului va trimite mesajul, individual, la toate procese care alcatuiesc grupul destinatie. O altă tehnică de adresare a membrilor grupului destinație difera total de toate cele mai sus, prin faptul că nu oferă transparenta. Dacă în toate tehnicile de mai sus, procesele utilizator care apelează primitivele de transmisie, nu trebuie sa cunoască componența grupului destinatie, în această tehnică, apelantul trebuie sa specifice o listă cu toate adresele componentelor grupului destinație (de exemplu, adresa IP); în acest caz, un parametru al primitivei send va fi un pointer la aceasta listă. O varianta a celor două tehnici prezentate mai sus este ca fiecare mesaj sa contina un predicat (expresie booleana), care să poată fi evaluat la destinatie; dacă valoarea obținuta este TRUE, mesajul este acceptat, altfel ignorat.

3.5.2. Primitive şi protocoale pentru comunicații de grup

Primitivele pentru comunicațiile de grup pot fi comune cu cele pentru comunicațiile punct la punct. Totusi, dacă mecanismul de comunicație punct la punct este RPC, acest lucru nu este posibil. Ca şi pentru comunicațiile punct la punct, primitivele pentru comunicațiile de grup pot fi cu şi fără blocare, cu şi fără bufferare, sigure şi nesigure.

În general, comunicațiile de grup trebuie sa furnizeze primitive pentru: (1) crearea şi distrugerea grupurilor; (2) inserarea unui nou membru într-un grup, eliminarea unui membru dintr-un grup; (3) atribuirea de pseudonume grupurilor, stergerea de pseudonume, cautarea unui grup după pseudonume; (4) transmiterea şi receptia de mesaje catre/de la procese apartinînd unui grup multicast. Trebuie mentionat că alaturarea sau parasirea la/unui grup trebuie facuta sincron cu transmisia de mesaje. Aceasta înseamnă că după eliminarea

BUPT

unui membru al unui grup, nu mai pot fi primite mesaje de la acest proces; analog, la alaturarea unui proces unui grup, el poate primi imediat mesaje.

O caracteristică a protocoalelor pentru comunicaţiile de grup este *atomicitatea*: mesajele sunt trimise unui grup, numai dacă ele sunt recepţionate corect de catre toţi membrii grupului; situaţii de genul în care numai unii membrii au receptionat un mesaj, iar ceilalti nu l-au receptionat (de exemplu datorita unei erori *overrun)* nu sunt permise. Urmatoarele două metode pot fi folosite pentru implementarea atomicităţii:

(1) la primirea mesajului, fiecare membru al grupului destinaţie va trimite o confirmare emiţătorului; dacă, după un interval de timeout, emiţătorul nu a primit toate confirmările, retransmite mesajul acolo unde este cazul. Metoda nu păstrează însă atomicitatea în cazul unui incident al nodului emiţător;

(2) o metoda care păstrează atomicitatea în orice situaţie a fost propusă de Joseph şi Birman [BJ87]: emiţătorul trimite mesajul, şi, după intervalul de timeout, îl retransmite acolo unde este necesar. Cînd un proces primeşte mesajul, verifică dacă este un mesaj nou: în caz afirmativ, îl transmite în continuare tuturor membrilor grupului (cu retransmisii, dacă este cazul), altfel ignora mesajul. Cu aceasta metoda, indiferent cate noduri sufera incidente sau cate mesaje se pierd, procesele care raman vor primi mesajul.

Pentru multe aplicaţii, protocolul multicast trebuie sa furnizeze unele garantii relativ la ordinea de livrare a mesajelor la destinatie. Aceste garanţii, prezentate în ordine crescanda a cerinţelor sunt [GS91][Tan93][Tan96]:

(1) Ordonarea mesajelor în cadrul unei singure surse: dacă mesajele m1 şi m2 sunt trimise de aceeaşi sursa la acelaşi grup multicast, atunci toate procesele grupului destinaţie trebuie sa primeasca mesajele în aceeaşi ordine.

(2) Ordonarea mesajelor în cazul mai multor surse: dacă mesajele m1 şi m2 sunt adresate catre acelasi grup multicast, atunci toate procesele grupului destinaţie trebuie sa primeasca mesajele în aceeaşi ordine (chiar dacă mesajele provin de la surse diferite).

(3) Ordonarea mesajelor în cazul unor grupuri nedisjuncte: analog cu (2), chiar dacă mesajele provin de la surse diferite şi chiar dacă sunt adresate unor grupuri distincte, dar care au procese comune.

Cerinţa (1) (rezolvată uneori de protoclul de comunicaţie) poate fi indeplinită dacă se ordonează (eventual numerotează) mesajele emiţătorului şi nu se permite recepţia mesajului decît în ordine crescatoare. Cerinţele (2) şi (3) sun mai greu de indeplinit. O soluţie este de a atribui o marcă de timp fiecărui mesaj şi de a livra mesajele în ordinea mărcilor de timp. Fie, de exemplu grupurile (fig.fig. 3.5.2.1): P={a, b, c, d) [i Q={c, d, e, f}. Presupunem că nodul X trimite un mesaj m1 cu marca de timp T1 catre grupul P. Cînd nodul C, de exemplu, recepţionează mesajul m1, nu îl poate livra imediat procesului destinatie, deoarece nu are certitudinea că mesajul are cea mai mică marca de timp faţă de alte mesaje nelivrate încă. Pentru indeplinirea cerinţei (2), nodul C trebuie sa compare marca de timp a mesajului cu marcile tuturor mesajelor trimise catre P (în cazul de mai sus, ale mesajelor de la Y). Pentru indeplinirea cerinţei (3), nodul C trebuie, în plus, sa compare marca de timp şi cu cele ale mesajelor trimise catre Q (deoarece nodul C face parte din două grupuri, P şi Q). Dacă potentialele surse de mesaje pentru C nu se cunosc, comparaţiile se fac relativ la mesajele tuturor nodurilor din sistem.

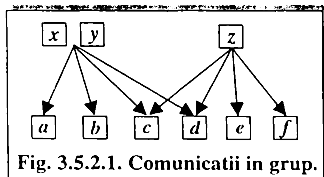


Fig. 3.5.2.1. Comunicatii in grup.

Birman şi Joseph [BJ87] au propus altă soluţie similara protocolului two-phase commit (vezi capitolul 7). Fiecare nod menţine cite o coada cu priorităţi pentru fiecare proces. Emiţătorul trimite mesajul catre destinatiile multicast, care îi atribuie un număr de prioritate propriu, cel mai mare pentru acel proces; mesajul este marcat *nelivrabil* şi memorat în coada. Fiecare receptor returnează emiţătorului numărul de prioritate; emiţătorul alege cel mai mare număr de prioritate şi îl trimite inapoi la receptori, care înlocuiesc vechiul număr de prioritate cu acesta şi marchează mesajul *livrabil.* Fiecare receptor va ordona în acest moment coada sa; cînd un mesaj din varful cozii devine *livrabil* el este transmis. Aceasta soluţie prezinta avantajul că asigură cerinţele (1), (2) şi (3) fără a necesita cunoasterea potentialelor surse de mesaje.

O soluţie centralizată este urmatoarea [CM84]: toate sursele transmit catre un nod central, care atribuie numere de secvenţă mesajelor şi le transmit destinatiei. Nodul central poate fi identificat printr-un token care circula prin sistem (din punct de vedere al transmisiei mesajelor, faptul că nodul central isi modifică în timp localizarea nu are importanta).

3.5.3. Exemple: MCL, ISIS,V-system, Amoeba

Se prezinta în cadrul acestei sectiuni trei exemple de comunicaţii de grup, care difera complet, atît din punct de vedere al conceptiei cît şi ca implementare. MCL şi ISIS se materializează sub forma unor toolkit-uri pentru comunicaţii de grup, pe cînd Amoeba implementează comunicaţiile de grup la nivelul nucleului.

BUPT

A. MCL

Un toolkit pentru comunicaţii de grup dezvoltat pentru sisiteme Unix este MCL; el cuprinde un set de 8 primitive de comunicaţii. Sistemul MCL este implementat prin intermediul a două procese daemon care se execută în fiecare nod: un proces daemon se ocupa cu transmisia mesajelor la alte noduri, iar celalalt numit daemon de distribuţie cu distributia mesajelor receptionate la procesele destinatie. În MCL, pentru a transmite mesaje multicast se foloseşte transmisia broadcast.

B. ISIS

Un alt toolkit, cunoscut şi mult referit în literatura de comunicaţii de grup, este sistemul ISIS (Universitatea Lornell, Ithaca, New York [BJ87][BJ90]); acesta constă într-un set de programe care se pot executa în sistemul de operare Unix şi în alte cateva sisteme de operare cunoscute. În ISIS sunt definite următoarele trei primitive pentru multicast [BJ90]: (1) Primitiva *GBCAST (group broadcast),* cea mai restrictiva, impune respectarea totală a conditiilor de ordonare a mesajelor prezentate la 3.5.2. Primitiva se foloseşte pentru gestiunea membrilor grupurilor, inclusiv în cazul unui membru care dispare în urma unei avarii grave a nodului sau. Argumentele primitivei sunt mesajul şi identificatorul grupului de procese, care este translatat în setul de adrese ale membrilor grupului, pe baza unei liste locale cu componenţa grupurilor, ce se menţine în fiecare nod. Deoarece execuţia primitivelor GBCAST păstrează ordinea mesajelor, rezultă că, deşi listele grupurilor nu sunt actualizate simultan, toti membrii grupurilor vor percepe aceeaşi secvenţă de modificare a grupurilor. (2) Primitiva *BCAST* este mai puţin restrictivă decît cea precedentă, fiind folosită pentru transmisia mesajelor obisnuite. Ea se utilizează în situatiile în care nu este necesară o ordonare totală a mesajelor multicast, în sensul că dacă două mesaje sunt primite într-o ordine la un nod, ele vor fi receptionate în aceeaşi ordine la toate destinatiile, dar nu neaparat în ordinea în care au fost emise. De exemplu, dacă un grup de procese se utilizează pentru a gestiona o structură tip coada, replicata, atunci BCAST se poate utiliza pentru a transmite operaţiile asupra cozii; deoarece ordinea operaţiilor va fi aceeaşi la toate nodurile, copiile cozii vor ramane consistente. Primitiva are sintaxa BCAST(msg, label, dest); msg şi dest reprezinta mesajul, respectiv destinatia iar label este un sir de caractere care specifică comportarea primitivei: numai primitivele BCAST care au aceeaşi valoare pentru label păstraza aceeaşi ordine de livrare a mesajelor la noduri. Pentru implementarea ambelor primitive GBCAST şi BCAST, ISIS foloseşte un protocol asemanator protocolului two-phase-commit, descris în sectiunea 3.5.2, completat pentru GBCAST cu un mecanism care elimina, mesajele provenind de la un proces membru disparut, înainte de a se livra un GBCAST care să anunte caderea procesului. (3) Primitiva OBCAST (ordered broadcast) este cea mai puţin restrictiva. Avînd sintaxa OBCAST(msg, dest) primitivele OBCAST asigură transmisia atomică a mesajelor şi ordonarea lor într-un nod numai în cazul unei *relaţii cauzale* între două primitive OBCAST. Relatia cauzală este referita aici în sensul Lamport (vezi sectiunea 4.1): două evenimente sunt în relaţie cauzală dacă comportarea celui de-al doilea poate fi influentată de primul; astfel, dacă nodul A trimite un mesaj la B şi apoi B analizînd mesajul primit trimite un alt mesaj la C, cele două operaţii de emisie sunt în relaţie cauzala; în caz contrar, de exemplu dacă A emite un mesaj la C şi, aproximativ în acelaşi timp, C emite un mesaj la D, evenimentele sunt concurente. Protocolul pentru implementarea primitivei OBCAST se poate descrie astfel: fiecărui proces membru al unui grup i se asociaza un vector avînd atatea componente cîti membri sunt în acel grup; componenta i a vectorului este numărul asociat ultimului mesaj primit de la membrul i (initial vectorul este setat la 0). Cînd un proces trimite un mesaj, se incrementează componenta corespunzatoare lui şi se trimite şi vectorul în cadrul mesajului. Fie V_i aceasta componentă şi L_i componenta corespunzatoare a vectorului unui receptor. Algoritmul decide cînd se livrează mesajul procesului sau se bufferează mesajul pentru a fi livrat mai tarziu. Figura 3.5.3.1 conţine un exemplu de ordonare a mesajelor în cazul a doua operaţii de emisie de mesaje pentru un grup format din 3 procese. Se observă din figură că, dacă presupunem că sursa unui mesaj este j, conditiile de livrare a mesajului sunt sunt: (1) $V_j = L_j+1$; aceasta impune deci ca mesajul primit sa fie într-adevar urmatorul în secvenţa de mesaje primite de la sursa j, altfel spus, sa nu fi fost un mesaj pierdut; (2) $V_i <> L_i$ pentru orice $i <> j$; aceasta impune că emiţătorul sa nu fi primit un mesaj pe care nodul receptor nu l-a primit.

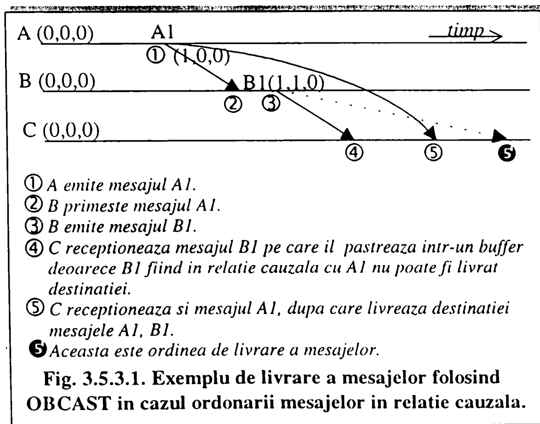


Fig. 3.5.3.1. Exemplu de livrare a mesajelor folosind OBCAST in cazul ordonarii mesajelor in relatie cauzala.

BUPT

În figura 3.5.3.5 se prezinta un exemplu de interacţiune între un client şi un grup de procese în care se folosesc toate tipurile de primitive.

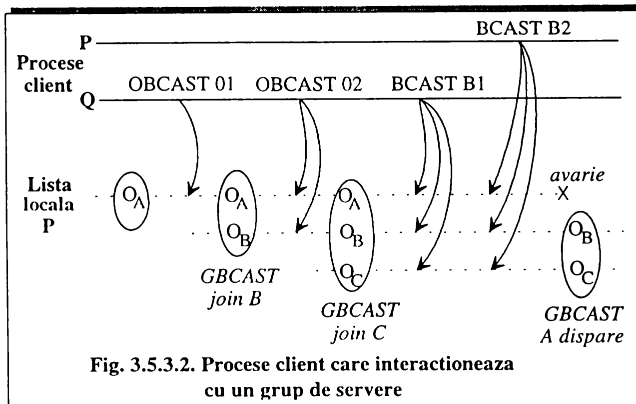


Fig. 3.5.3.2. Procese client care interactioneaza cu un grup de servere

C.Amoeba

În sistemul de operare Amoeba, alături de forme de baza de comunicaţie RPC, sunt prevazute şi comunicaţii de grup [Tan93][Tan96]. În Amoeba grupurile sunt inchise; singura formă de acces a unui client la serviciul furnizat de un grup este ca clientul sa execute un apel de procedură la unul din serverele grupului; dacă este necesar, membrii grupului pot comunica, în continuare între ei, folosind primitive pentru comunicaţii în grup, pentru a stabili cine execută serviciul cerut. Primitivele pentru comunicaţii în grup în Amoeba sunt prezentate în fig.3.5.3.3. Se asigură o transmisie sigura, atomică a mesajelor catre membrii unui grup, chiar şi în situaţii de mesaje pierdute sau depăşirea capacităţii bufferelor. Amoeba implementează pentru comunicaţiile în grup un protocol care utilizează metoda centralizată prezentată la sfarsitul sectiunii 3.5.2.

Primitiva	Semnificatie
Create Group	Creaza un grup nou şi seteaza anumiti parametri, ca de exemplu, numarul maxim de membri disparuti ai unui grup pentru care grupul continua sa lucreze
Join Group	Permite introducerea unor membri noi într-un grup
Leave Group	Permite unui membru parasirea unui grup
Send To Group	Transmite singur un mesaj membrilor unui grup
Receive From Group	Permite receptionarea unui mesaj de la un grup; blocare, daca nu este disponibil nici un mesaj
Reset Group	Se utilizeaza la revenirea după o avarie grava şi specifica numarul minim de membri necesari pentru a forma un nou grup; daca nucleul reuseste sa contacteze numărul respectiv de membri grupul se reconstruieste

Fig. 3.5.3.3. Primitive pentru comunicaţia de grup în Amoeba.

BUPT

Capitolul 4.

SINCRONIZAREA ÎN SISTEME DE OPERARE DISTRIBUITE

Termenul *sincronizare* este folosit în sisteme distribuite pentru a referi trei probleme distincte, dar înrudite: (1) sincronizare între emițătorul și receptorul unui mesaj; (2) specificarea și controlul acțiunilor unor procese care cooperează în cadrul unei aplicații distribuite; (3) serializarea accesului concurent la obiectele partajate între mai multe procese; această problemă este cunoscută și în bazele de date sub denumirea de *controlul concurentei*. În capitolul de față, se vor trata aspecte privitoare numai la problemele de tipul (2); problema (1) a fost tratată în capitolul precedent, prezentîndu-se în secțiunea 3.2.2. patru moduri posibile de sincronizare a unui receptor cu un emițător, (în cadrul operației *send*), iar aspecte referitoare la controlul concurenței vor fi prezentate în contextul unui sistem de obiecte distribuite în cap. 7.

O concluzie a acestui capitol va fi aceea că sincronizarea în sisteme distribuite implicit întotdeauna și comunicare; *sincronizarea nu se poate realiza fără comunicare.*

O prima definiție pentru sincronizare, avînd în vedere cele de mai sus, este următoarea:

Definitie:

Sincronizarea este o ordonare temporală a unui set de evenimente produse în timp de două sau mai multe procese concurente.

Datorită distribuirii resurselor, întîrzierilor de transmisie, lipsa unei memorii comune și deci lipsa unor informații globale, algoritmii și metodele pentru sincronizare din sistemele centralizate nu pot fi utilizate în sistemele distribuite.

Mecanismele de sincronizare pentru sistemele distribuite vor trebui să determine, printr-o metodă oarecare, o ordonare totală sau parțială a unui set de evenimente. Capitolul de față va prezenta mai întîi aspecte ale ordonării evenimentelor într-un sistem distribuit și apoi, pe baza acestora, două clase mari de mecanisme de sincronizare: *centralizate* și *distribuite*. Unele dintre aceste mecanisme se bazează pe conceptul de *excludere reciprocă* care presupune o ordonare a unor secvențe de cod executate de procesele concurente; aceste mecanisme pot utiliza diferite tehnici precum: *un proces central coordonator* sau un *token circulant*. Alte mecanisme se bazează pe înregistrarea evenimentelor semnificative în cursul unor prelucrări asincrone; acestea utilizează obiecte precum *contori de evenimente* și *secvențiatori*, sau *semafoare distribuite*.

4. 1. Ordonarea evenimentelor în sisteme distribuite

În sistemele centralizate este întotdeauna posibil să se determine ordinea în care au loc două evenimente deoarece există o memorie și un ceas comun tuturor proceselor. În sistemele distribuite, neexistînd nici memorie comună și, în general, nici un ceas global, comun, este uneori imposibil să se distingă că un eveniment a avut loc înaintea altuia. O soluție de a ordona evenimentele, în sisteme distribuite , fără a utiliza ceasuri fizice, se bazează pe relația *happend-before (s-a întamplat înainte)*. Aceasta relație este o relație de ordine parțială pentru evenimentele sistemului distribuit, dedusă din legea cauzalității ([BBK91][Tan96][Gos91]).

Deoarece execuția unui proces se face secvențial, toate evenimentele acelui proces vor fi total ordonate. De asemenea, conform legii cauzalității, un mesaj poate fi recepționat numai după ce a fost emis. Prin urmare, relația *happend-before*, notată cu < ', poate fi definită peste un set de evenimente astfel (presupunînd că emisia și recepția unui mesaj constituie un eveniment): (1) Daca A și B sunt evenimente în același proces și A a fost executat înaintea lui B, atunci A< ' B; (2) Daca A este un eveniment de emisie al unui mesaj într-un proces și B este un eveniment de recepție al acelui mesaj de catre un alt proces, atunci A< ' B; (3) Daca A< ' B și B< ' C atunci A< ' C;

Deoarece un eveniment nu se poate precede pe el însuși, rezultă că relația "< '" este o relație nereflexivă, de *ordine parțială*. Daca două evenimente A și B nu sunt în relație < ' (adică A nu l-a precedat pe B și nici B nu l-a precedat pe A) atunci cele două evenimente sunt concurente: nici unul din aceste evenimente nu îl poate afecta, cauzal, pe celălalt. Relația happend-before este reprezentată printr-un exemplu, utilizînd o diagramă timp-spațiu în fig. 4.1.1. Evenimentele p1, q1, r0 corespund unor emisii de mesaje iar p4, q4, r1 unor

BUPT

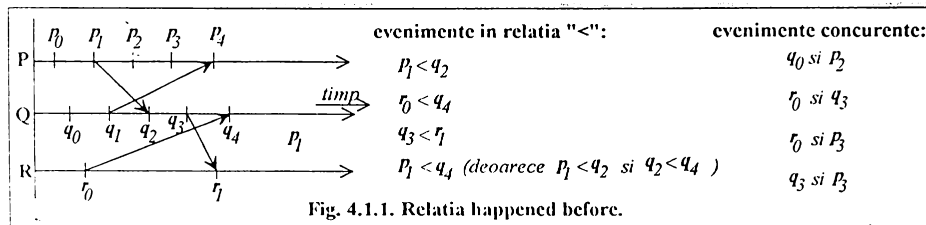


Fig. 4.1.1. Relatia happened before.

recepţii de mesaje; se observă că două evenimente A şi B sunt *concurente* dacă şi numai dacă nu există o cale de la A la B sau de la B la A.

4.2. Soluţii centralizate pentru sincronizare

Soluţiile centralizate pentru sincronizare desemnează o entitate unică a sistemului distibuit ca entitate coordonatoare şi, implicit, nodul pe care se află aceasta, ca nod coordonator. O astfel de entitate poate fi un ceas fizic global sau un proces coordonator unic în sistem.

Definiţie:

O entitate de sincronizare se spune că este *coordonatoare* sau *centrală* dacă: (1) posedă un nume unic, cunoscut de toate procesele care o folosesc pentru sincronizare; (2) în orice moment ea poate fi accesată de oricare proces.

Un algoritm centralizat este caracterizat prin două proprietăţi: (1) deciziile se iau numai la nivelul nodului coordonator; (2) toate informaţiile necesare deciziilor sunt concentrate în nodul coordonator.

Dezavantajele algoritmilor centralizaţi constau în vulnerabilitatea lor ca urmare a defectării nodului coordonator şi în circulaţia intensă de mesaje în jurul nodului coordonator.

Unele sisteme, proiectate cu toleranţă la defecte, furnizează, printr-o tehnică specială de recuperare, posibilitatea înlocuirii entităţii coordonatoare (există algoritmi specifici de alegere a unui coordonator pentru aceste cazuri, vezi subcapitolul 4.4). Între două avarii grave, care au ca urmare pierderea coordonatorului şi alegerea altuia, definiţia de mai sus ramîne valabilă.

4.2.1. Utilizarea unui ceas fizic global

Soluţia utilizării unui *ceas fizic global* presupune utilizarea unui ceas fizic unic în sistemul distribuit, analog sistemelor centralizate. Pentru a ordona evenimentele, procesele construiesc mărcile de timp utilizînd ceasul fizic global. Simplă, această soluţie are numeroase dezavantaje ([Gos91]):

(1) valoarea corectă a mărcilor de timp depinde în întregime de posibilitatea recepţionării corecte, în toate momentele, a valorii ceasului fizic;

(2) erorile de transmisie pot altera ordinea evenimentelor;

(3) este necesară estimarea apriori sau posteriori a întîrzierilor de transmisie; (4) gradul de perfecţiune al ceasului influenţează constrîngerile care se pot respecta în sistem.

4.2.2. Utilizarea unui proces central

Soluţia utilizează un proces unic în sistemul distribuit, coordonator, care recepţionează cererile de sincronizare ale tuturor proceselor din sistem. Sincronizarea care se poate obţine astfel este de tipul excludere reciprocă şi necesită trei mesaje: două pentru intrarea într-o secţiune critică şi unul pentru ieşire, astfel: (1) procesul care cere intrarea într-o secţiune critică trimite o cerere coordonatorului; (2) coordonatorul verifică dacă sunt alte procese în secţiunea critică. În caz negativ, trimite un mesaj de răspuns, de acceptare a intrării în secţiunea critica; în caz afirmativ, cererea este introdusă într-o coadă de aşteptare; (3) la primirea mesajului de răspuns, procesul intră în secţiunea critică; (4) la ieşirea din secţiunea critică, procesul trimite coordonatorului un mesaj de terminare secţiune critică; (5) la primirea mesajului, coordonatorul preia din coada de aşteptare următoarea cerere de intrare într-o secţiune critică şi trimite un mesaj de răspuns acelui proces care a lansat cererea.

Utilizînd o planificare FIFO, nu poate avea loc amînarea nedefinită a proceselor la intrarea în secţiunea critică. Soluţia are dezavantajul că necesită trei mesaje pentru o secţiune critică şi că, în cazul dispariţiei coordonatorului, trebuie aplicat un algoritm de alegere al unui nou coordonator; după desemnare, acesta trebuie să interogheze toate procesele pentru a reconstitui coada de aşteptare.

În ceea ce priveşte utilizarea soluţiei procesului central pot fi date multe exemple. În bazele de date distribuite, de exemplu, controlul concurenţei se poate baza pe un *proces unic*, care primeşte toate cererile proceselor ce iniţiază acţiuni; sincronizarea acţiunilor este realizată de procesul unic. Un alt exemplu este

BUPT

algoritmul centralizat pentru *detecția interblocărilor* în sistemele distribuite; acesta utilizează un proces central care construiește un graf global al conflictelor (vezi subcapitolul 7.5.3.3.).

4.2.3. Contori de evenimente și secvențiatori

Coordonarea efectivă a proceselor poate fi atinsă dacă se menține o relație de ordine parțială pentru toate evenimentele proceselor, mai precis o relație de ordine peste toate evenimentele înrudite. Acest lucru se realizează cu ajutorul *contorilor de evenimente.*

Definiție:

Se numește *contor de evenimente* o variabilă de tip întreg, inițial cu valoarea 0 și care ia valori în ordine strict crescătoare.

Un contor de evenimente se manipulează prin urmatoarele *operații*: (1) *advance.* Operația *advance(E)* se utilizează pentru a anunța apariția unui eveniment legat de E, și provoacă incrementarea cu 1 a lui E. (2) *read:* Operația *read(E)* se folosește de catre un proces pentru a returna valoarea curentă a lui E; (3) *await.* Operația *await(E,v)* provoacă suspendarea procesului apelant atîta timp cît E<=v.

Un contor de evenimente este analog cu un *ceas global* care înaintează la fiecare apariție a unui eveniment semnificativ, prin execuția explicită a operației *advance*, care apare astfel ca un eveniment de semnalizare. Un proces se sincronizează cu *ceasul global* executînd operația *await*, el ramîne astfel blocat pînă cînd *ceasul global* atinge un timp predefinit, sau, altfel spus, pînă cînd se execută v apeluri *advance.* Cînd sincronizarea este mai complexă, se poate folosi și operația *read* care returnează numarul de apeluri *advance* executate. Aceste primitive permit rezolvarea multor probleme de sincronizare, clasice în sistemele de operare: problema producator-consumator ([Gos91], [Nut92]), problema readers-writers ([Gos91]).

Reed și Kanodia au arătat pentru prima oară ([Gos91] [RK79], [Lam83]) cum contorii de evenimente pot fi extinși la sisteme distribuite. Ei au considerat că execuția primitivelor *advance, read* și *wait* constituie evenimente și au extins relația de ordine parțială <' a lui Lamport (prezentată în 4.1) astfel ([Gos91]):

(1) Dacă A și B sunt evenimente în același proces, și A a fost executat înaintea lui B, atunci A<' B;

(2) Dacă A este un eveniment de emisie al unui mesaj dintr-un proces și B este evenimentul de recepție al aceluiași mesaj, atunci A<' B;

(3) Dacă A<' B și C<' D atunci A<' D sau C<' B .

(4) Pentru toate evenimentele A, A<' A nu este adevarată.

Ca și în modelul lui Lamport, două procese A și B sunt concurente dacă relațiile A<'B și B<' A nu sunt adevărate. Relația impusă de cele patru condiții de mai sus a fost notată => și definită astfel: *a "precede" pe b (a => b) daca, și numai dacă, nu este adevărată relația b <' a.* Pe baza ei, Reed și Kanodia au specificat dependențele de timp dintre primitive astfel. Există posibilitatea ca aceste operații, aplicate asupra aceluiași contor să fie concurente. Dependențele de timp sunt:

(5) Dacă W_E este execuția unei operații *await(E, t)* atunci trebuie să existe cel puțin t membri ai mulțimii {A_E:A_E este execuția unei operații *advance(E)* și A_E=>W_E};

(6) Dacă R_E este execuția unei operații *read*, v= *read(E)*, atunci există cel mult v membri ai mulțimii {A_t, A_t este execuția operației *advance(E)* și A_E<' R_E} și există cel puțin v membri ai mulțimii {A_t, A_t este execuția operației *advance*(A) și A_E=> R_E}.

Prin *contor de evenimente singular* se desemnează un contor de evenimente care lucrează corect dacă nu există operații *advance* concurente. Într-un sistem distribuit, un contor de evenimente poate fi construit pe baza mai multor contoare de evenimente singulare, fiecare situat pe cîte un nod. Numai procesele locale corespondente contoarelor singulare le pot modifica. Aceasta înseamnă că o operație *advance* asupra contorului global va avansa contorul local singular. O operație *read* asupra contorului global se realizează prin citirea contoarelor de evenimente singulare, în orice ordine și însumînd valorile rezultate. Procesele care execută *advance* sunt denumite *semnalizatori*, iar cei care execută *read observatori*.

Contorii de evenimente stabilesc o relație de ordine parțială peste evenimentele legate ca semnificație de contor, dar nu pot fi utilizați pentru evenimente neînrudite. În astfel de cazuri se utilizează *secvențiatorii*. Un secvențiator poate fi utilizat pentru a stabili o relație de ordine totală peste evenimentele legate de destinație.

Definiție:

Se numește *secvențiator* o variabilă de tip întreg cu valoare inițială 0, care ia numai valori strict crescătoare.

O singura primitivă este permisă asupra unui secvențiator: primitiva *ticket*. Operația *ticket(T)* returnează valoarea curentă a secvențiatorului și incrementează cu 1 valoarea lui. Operația ticket este *atomică*.

În fig. 4.2.3.1. se prezintă ca exemplu *problema accesului în citire și actualizare a unei baze de date*, rezolvată cu contori de evenimente și secvențiatori. Procesele care accesează baza de date pentru citire au rol numai de observator în ceea ce privește contorii de evenimente asociați bazei de date; aceasta înseamnă că nu pot executa operații *advance* asupra acestora și operația *ticket*.

```
Exclusive Acces()                              Shared Acces()
{                                              {
 /* cand E1 > E2 baza de date este */            int i;
 /*    accesata in acces exclusiv   */         retry:
 int i;                                          S1 <- read (E1);
 advance (E1);                                   await (E2, j);
i <- ticket (T); /* secventiator folosit pentru    /* asteptare pentru acces partajat *
          excluderea reciproca a proceselor        /* Acces partajat in baza de date */
               care cer acces exclusiv */        S2 <- read (E1);
 await (E2, i);                                  if ( S1!=S2 )
 /* asteptare pentru acces exclusiv */              goto retry; /* date inconsistente*/
/*   Acces exclusiv in baza de date   */         }
advance (E2);
}
```

Fig. 4.2.3.1. Utilizarea indicatorilor de eveniment si a secventatorilor in problema accesului exclusiv si partajat la o baza de date

Soluția Read și Kanodia de implementare a contorilor de evenimente se bazează pe presupunerea că în sistemul distribuit se acceptă erori în comunicație, dar nu și avarii grave ale nodurilor. Bullis și Franta [BF80] au propus o soluție valabilă în rețelele care admit transmisii broadcast și respectă urmatoarele cerințe: (1) comunicațiile nesigure, mesajele pot fi pierdute, duplicate sau modificate; (2) nodurile pot suferi avarii grave, iar informațiile deținute de acestea pierdute complet; (3) avaria unui nod poate fi identificată de celelalte noduri, în situația în care un proces de pe acel nod nu mai raspunde la mesaje. Scopul implementării Bullis și Franta este, pe de o parte, de a asigura funcționarea sistemului chiar și în cazul dispariției unui nod, și pe de altă parte, de a pennite revenirea unui nod în sistem după înlăturarea unei avarii. Bullis și Franta au propus ca fiecare nod să conțină o copie a fiecarui contor de eveniment. Aceasta implică transmiterea unui mesaj de broadcast la fiecare operație *advance* a unui contor de evenimente, mesaj care să fie primit de toate nodurile, pentru a asigura consistența contorilor.

Primitiva *await* a fost modificată la sintaxa: *await (E, v, timeout)* pentru a pennite deblocarea unui proces care așteapta nedefinit. Semantica noii primitive *await* este:

așteapta pana E>=v *sau* timeout
dacă E>=v *atunci* returnează 0
altfel returnează -1

```
Exclusive Access ()                      Shared Acces ()
{                                        {
 /* cand E1>=E2 baza de date este */     retry:
 /* accesata in mod exclusiv */             S1<--lread (E1);
int i;                                      eroare<--await (E2,S1,timeout);
advance (E1);                               /*o eroare semnifica un proces in
t<--ticket (T);                             acces exclusiv ramas in sectiunea critica
c<--lread (E2);                             */
for ( i=c; i<t; i++ ) {                     if ( eroare != -1 ) {
  j<--await (E2,i,timeout);                 /*
  if ( j= -1 ) return ('eroare');            Acces partajat in baza de date
  /* o eroare semnifica un proces in         citire din baza
     acces exclusiv ramas in                */
     sectiunea critica                      }
    */                                      S2<--max (rread (E1,1),...,rread (E1,N));
}                                           if ( ( S2 != -1 ) && ( eroare != -1 ) ) {
/*                                               if ( S1= =S2 ) return
 Acces exclusiv in baza de date                  else
 citire si scriere in baza                       /* baza de date ar putea
*/                                              fi utilizata in timpul citirii
advance (E2);                                    */
}                                               goto retry;
                                               }
                                              else
                                             /* daca S2 = -1 toate nodurile
                                              cu acces exclusiv nu mai raspund
                                             */
                                            /* baza de date este inconsistenta */
                                             return 'eroare'
                                         }
```

Fig. 4.2.3.2 Algoritmul Bullis si Franta

BUPT

Primitiva *read* a fost înlocuită cu alte două primitive: *lread* (local read) şi *rread* (remote read). Primitiva *lread (E)* returnează valoarea copiei locale a lui E; primitiva *rread (E, n)* returnează valoarea copiei contorului de pe nodul n sau -1 în cazul cînd nodul nu raspunde. În fig. 4.2.3.2. se prezintă acelaşi exemplu, al accesului exclusiv şi partajat la o baza de date, utilizînd implementarea Bullis şi Franta a contorilor.

4.3. Soluţii distribuite pentru sincronizare

Un *algoritm complet distribuit* este caracterizat de următoarele proprietăţi; (1) toate nodurile deţin şi menţin aceeaşi cantitate de informaţie; (2) toate nodurile iau decizii numai pe baza informaţiilor locale; (3) toate nodurile au responsabilităţi egale în raport cu decizia finală; (4) toate nodurile depun acelaşi efort în realizarea deciziei finale; (5) avaria unui nod nu conduce în general la compromitarea întregului sistem.

În practică, toate aceste cinci proprietăţi sunt respectate în totalitate de puţini algoritmi distribuiţi. Unii algoritmi, de exemplu, pot cere ca fiecare nod să aibă aceeaşi responsabilitate în decizia finală, dar efortul depus de fiecare nod pentru obţinerea excluderii reciproce să nu fie acelaşi; alţi algoritmi pot conduce la compromiterea sistemului, chiar cînd un singur nod suferă o avarie gravă.

Indiferent de modul şi de gradul de distribuire, există două presupuneri fundamentale, comune tuturor algoritmilor distribuiţi:

Ipoteza 1:. Fiecare nod are numai o vedere parţială a întrgului sistem, pe care trebuie să-şi bazeze deciziile;

Ipoteza 2: Nu există un ceas fizic global.

Ultima presupunere este o constrîngere majoră, avînd în vedere că ordonarea temporală a evenimentelor este un concept fundamental din punct de vedere al sincronizării lor. A spune că într-un sistem distribuit un eveniment A a avut loc înainte de alt eveniment B, nu poate fi o afirmaţie sigură din două motive: (1) poate exista o întîrziere de transmisie între momentul în care are loc un eveniment şi momentul în care este observat; (2) dacă fiecare nod conţine un ceas fizic, diferenţa dintre ele coduce la variaţii la citirea timpului.

Pentru a construi baza de dezvoltare a algoritmilor distribuiţi, trebuie stabilită o metodă de ordonare a evenimentelor într-un sistem distribuit.

4.3.1. Sincronizarea ceasurilor

1.Ceasuri fizice

S-a aratat în secţiunea 4.2.1. că utilizarea unui ceas fizic unic, global într-un sistem distribuit prezintă o serie de dezavantaje, dintre care cel mai important este necesitatea aprecierii întîrzierilor de transmisie. Dacă se utilizează cîte un ceas fizic în fiecare nod, atunci ele trebuie sincronizate astfel încît diferenţa relativă a oricaror două ceasuri fizice să se păstreze sub o limită constantă fixată. Deoarece două ceasuri nu vor funcţiona niciodată cu aceeaşi rată, diferenţa dintre ele crescînd mereu, Lamport ([Lam78][BBK91][Gos91]) a prezentat o soluţie a problemei sincronizării ceasurilor, prin introducerea unui algoritm, pe baza a două condiţii şi a două reguli.

Prima condiţie descrie funcţionarea corectă a unui ceas fizic, astfel:

Condiţia 1: Există o constantă $k<<1$, astfel încît pentru toate ceasurile i: $|dC_i/dt - 1| < k$, unde $C_i(t)$ este o funcţie continuă, diferenţiabilă, iar dC_i/dt reprezintă rata cu care funcţionează ceasul i la momentul t.

În plus, toate ceasurile trebuie sincronizate astfel încît $C_i(t) \approx C_j(t)$, pentru orice i, j, t , astfel:

Condiţia 2: Există o constantă suficient de mică astfel încît pentru toti i, j, $|C_i(t) - C_j(t)| < \varepsilon$.

Deoarece întotdeauna valorile a două ceasuri tind să se îndepărteze, este necesar un algoritm care sa asigure respectarea condiţiei 2. Pentru a dezvolta un astfel de algoritm, se presupune că procesul P_i trimite un mesaj m la timpul t; mesajul este primit de P' la momentul t', deci întîrzierea de transmisie este $\delta_m = t' - t$. Procesul destinaţie cunoaşte numai întîrzierea minimă $\mu_m >= 0$ ($\mu_m <= \delta_m$). Diferenţa dintre aceste două întîrzieri se numeşte *diferenţă nepredictibilă* $\rho_m = \delta_m - \mu_m$. Pot fi enumerate acum regulile pentru ceasurile fizice:

Regula 1: Pentru fiecare i, dacă P_i nu recepţionează un mesaj la momentul t, atunci C_i este diferenţiabilă la t şi $dC_i(t)/dt>0$.

Regula 2:

(a) Dacă P_i trimite un mesaj m la momentul t, atunci m conţine o marca de timp $TS_m = C_i(t)$.

(b) După primirea mesajului m la momentul t', procesul P_j stabileşte $C_j(t')$ egal cu max ($C_j(t'-0)$, $TS_m + \mu_m$).

Un proces trebuie deci să cunoască numai ceasul său fizic şi marca de timp a mesajelor recepţionate. Trebuie arătat că algoritmul prezentat prin cele două reguli de mai sus respectă condiţia 2. Presupunem că sistemul este modelat printr-un graf de procese puternic conectat; arcele vor reprezenta liniile de comunicaţie pe care pot fi trimise direct mesaje între procese. *Diametrul* acestui graf este cel mai mic număr d, astfel încît,

BUPT

pentru orice pereche de procese P_a, P_b există o cale de la P_a la P_b avînd cel mult d arce. Vom spune că, pe un arc, se trimite un mesaj de la P_i la P_j la fiecare τ secunde, dacă pentru orice t, P_i trimite cel puțin un mesaj la P_j între momentele t, $t+\tau$. La fiecare τ secunde se trimite un mesaj de sincronizare pe fiecare arc, care conține o marcă de timp TS. După primirea mesajului, dacă este nevoie, un proces va avansa ceasul său local la valoarea conținută în mesaj. Se presupune că întîrzierile de transmisie sunt limitate de o limită inferioară u și una superioară u+z, cunoscute. Fie K marja de eroare a ceasului, K<10E-6 și ε diferența permisă pentru două ceasuri. Dacă $\varepsilon/(1-K)<=u$ și $\varepsilon << \tau$ atunci este posibil să se aprecieze valoarea aproximativă pentru ε ca fiind $d(2K\tau+z)$.

În final trebuie precizat că, în funcție de cerințele legate de diferența dintre ceasuri și de validitatea presupunerilor legate de limitele întîrzierilor, se poate decide că, fie se poate asuma riscul pierderii unor mesaje de sincronizare din cînd în cînd, datorită numarului excesiv de mare de mesaje care trebuie transmise, obținîndu-se astfel o sincronizare numită *probabilistică*, fie nu se poate asuma acest risc; parametrul cheie este raportul z/u.

Un algoritm de sincronizare a ceasurilor fizice ([Tan96]) se bazează pe divizarea timpului în intervale fixe de resincronizare și pe transmiterea de mesaje broadcast ce conțin timpul curent de catre fiecare nod la începutul fiecarui interval $[T_0+iR, T_0+(i+1)R]$ (T_0 este un moment în trecut stabilit de comun acord, iar R un parametru de sistem). După ce un nod a transmis mesajul broadcast, startează un timer local pentru a colecta toate celelalte mesaje în timpul unui interval S. După ce s-au primit toate mesajele se calculează timpul curent prin media valorilor primite în mesaje, eventual eliminînd cele mai mari respectiv, mici *m* valori.

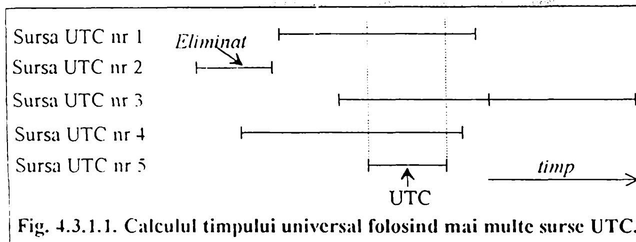


Fig. 4.3.1.1. Calculul timpului universal folosind mai multe surse UTC.

Pentru sisteme distribuite în care se cere o sincronizare cît mai perfectă cu timpul universal, UTC (Universal Coordinated Time) pot fi folosite mai multe surse UTC. Pentru sincronizarea acestora se prezintă ca exemplu algoritmul utilizat în DCE: periodic, la începutul unui interval de timp fixat, (de exemplu 1 min) fiecare procesor dotat cu o sursă UTC transmite un mesaj broadcast către toate nodurile. Cînd un procesor primește mesaje de la toate nodurile cu surse UTC, elimină întîi specificatiile disjuncte (figura 4.3.1.1) și apoi calculează timpul curent ca mijlocul intersecției tuturor specificațiilor.

2. Ceasuri logice

În multe aplicații, este suficient ca toate calculatoarele să perceapă același timp, nefiind necesar ca acesta să coincidă cu timpul real. Pentru aceasta clasă de aplicații este importantă consistența ceasurilor și nu dacă au sau nu valoarea timpului real; de aceea se folosesc *ceasurile logice*, care nu implică prezența unui ceas fizic. Ceasurile logice permit extinderea relației *happend-before* la o relație totală de ordine pe evenimentele unui sistem distribuit ([Lam78]). Cerințele unei astfel de relații totale sunt: pentru orice pereche de evenimente A și B, dacă A<'B atunci marca de timp a lui A este mai mică decat marca de timp a lui B (reciproca nu este valabilă).

Un ceas logic LC_i se asociază unui proces P_i; el este definit ca o funcție care asociază un numar întreg nenegativ LC_i <a> fiecărui eveniment *a* din proces. Întregul sistem de ceasuri logice se reprezintă printr-o funcție LC care atribuie oricarui eveniment *b* numărul C<b> unde C<b> = C_i<b> dacă *b* este un eveniment în procesul P_i. Pe baza definiției poate fi enunțată acum condiția de ceas logic a lui Lamport:

Definiție.

Fie C o funcție care asignează o valoare nenegativa fiecărui eveniment e_i; pentru evenimentele e_i, e_j avem atunci: $e_i <' e_j \Rightarrow LC(e_i) < LC(e_j)$

Rezultă că ceasul logic poate fi implementat printr-un contor care este incrementat intre oricare două evenimente succesive dintr-un proces. Deoarece un contor furnizează valori crescatoare, el asignează valori unice fiecarui eveniment iar dacă un eveniment *a* precedă într-un proces P_i un eveniment *b*, atunci LC(a)< LC(b). Marca de timp asociată unui eveniment este valoarea contorului (ceasul logic) pentru acel eveniment. Această schema asigură, într-adevar cerințele relației de ordine totală, dar numai pentru evenimente ale aceluiași proces; ea trebuie completată avînd în vedere situația proceselor care-și comunică mesaje.

Fie $\{p_0, p_1, \dots p_{n-1}\}$ un set de procese, fiecare cu un numar de evenimente și fie LC_i ceasul logic corespunzător procesului *i*, care asignează un număr întreg nenegativ evenimentelor procesului. Condițiile următoare asigură respectarea globală a condiției de ceas logic a lui Lamport:

(1) Dacă evenimentele e_j, e_k au loc în acelasi proces și e_j are loc înaintea lui e_k, atunci: $LC_i(e_j) < LC_i(e_k)$

BUPT

(2) Dacă e_s este un eveniment de emisie a unui mesaj de catre procesul P_i, iar e_r este un eveniment de recepție a acestui mesaj de catre procesul P_j, atunci: $LC_i(e_s) < LC_j(e_r)$

Pe baza acestor condiții se poate defini complet modul de implementare al ceasului logic printr-un contor. Fie LC_i contorul asociat procesului P_i; el devine un ceas logic dacă:

(1) C_i este incrementat între două evenimente succesive ale procesului

(2) Dacă evenimentul e_r este o emisie unicast sau multicast a mesajului m de catre procesul P_i, atunci mesajul i va include marca de timp a evenimentului TS_m (valoarea ceasului logic). Cînd procesul P_j recepționează mesajul m, setează ceasul sau logic LC_j la valoarea $\max(TS_m+1, LC_j)$.

Condiția (2) de mai sus asigură executarea corecției de timp numai prin adăugarea de valori pozitive ceasurilor, niciodată prin scădere. În acest caz, ceasul unui proces P_j va fi sincronizat cu ceasul procesului P_i cînd se transmite un mesaj de la P_i la P_j. Figura 4.3.1.2. reprezintă succesiunea valorilor ceasurilor logice pentru trei procese. Diagrama din fig. 4.3.1.4. reprezintă progresul logic a trei procese aratate în fig. 4.3.1.3.,

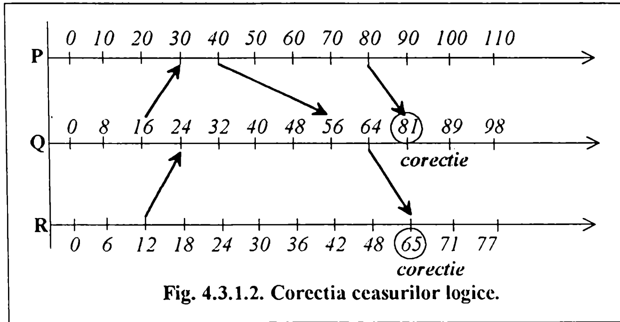


Fig. 4.3.1.2. Corectia ceasurilor logice.

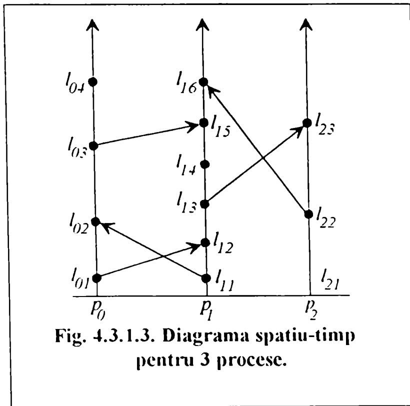


Fig. 4.3.1.3. Diagrama spatiu-timp pentru 3 procese.

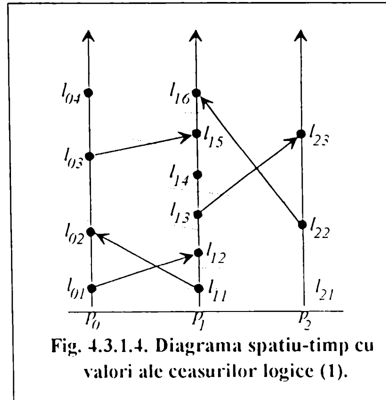


Fig. 4.3.1.4. Diagrama spatiu-timp cu valori ale ceasurilor logice (1).

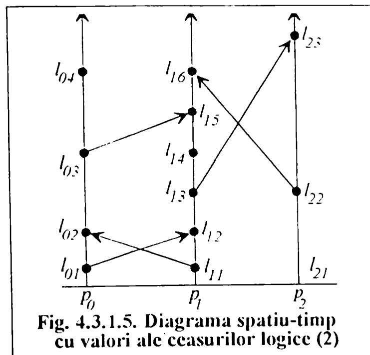


Fig. 4.3.1.5. Diagrama spatiu-timp cu valori ale ceasurilor logice (2)

BUPT

sugerînd o viziune consistentă a relației de ordine a evenimentelor în sistem; fig. 4.3.1.5. reprezintă de asemenea o viziune consistentă, de aceasta dată sugerînd că ceasurile locale sunt sincronizate în timp.

În final, pentru a realiza complet o relație de ordine totală, mai trebuie specificat că, în cazul a două evenimente concurente ordonarea trebuie facută ținînd cont de identitatea proceselor (procesele se ordonează după un criteriu oarecare, de exemplu în ordinea identificatorilor lor).

4.3.2. Algoritmi distribuiți pentru excludere reciprocă

Deși diferă prin cerințe și obiective, algoritmii distribuiți pentru excludere mutuală (Lamport [Gos91][Lam78], Ricard și Agrawala [RA81], Maekawa [MOO87]) au comune următoarele cerințe și caracteristici:

(1) Sistemul distribuit este văzut ca un sistem cu N noduri, numerotate de la 1 la N. Fiecare nod conține un proces care cere acces exclusiv la un obiect partajat. Cererea sa este comunicată și celorlalte procese.

(2) Proprietatea *pipeline* este adevărată; aceasta înseamnă că mesajele trimise de la un proces la altul sunt numerotate în ordinea transmiterii lor.

(3) Fiecare mesaj este livrat corect la destinație într-un timp finit;

4) Rețeaua este complet conectată; aceasta înseamnă că fiecare proces poate trimite mesaje direct la oricare alt proces.

Cerințele 2 și 3 se realizează prin metode descrise în capitolul 2.

În acest model al sistemului distribuit, fiecare proces, în afară de faptul că cere un acces exclusiv la obiectul partajat, servește și ca arbitru pentru a rezolva cererile care se suprapun în timp. Pentru a arbitra cererile, orice pereche de două cereri trebuie să fie cunoscută de cel puțin unul dintre arbitrii. Să presupunem, de exemplu, că pentru a obține acces exclusiv la obiect procesul i trebuie să obțină permisiunea fiecărui membru al unui subset S_i de procese din sistem; atunci, pentru a asigura excluderea reciprocă a proceselor i și j, S_i și S_j, trebuie să conțină un arbitru comun. Formal, acest lucru se exprimă astfel:

(a) *Proprietatea intersecției nenule* : $S_i \cap S_j \neq 0$ pentru orice combinație i, j cu $1 \leq i, j \leq N$. Dacă se respectă această proprietate, excluderea reciprocă este garantată dacă orice proces care cere acces la obiectul partajat obține permisiunea de la fiecare membru al lui S_i. Această proprietate este condiția necesară pentru realizarea excluderii reciproce. În plus, uneori sunt valabile și următoarele proprietăți:

(b) *Regula Efortului Egal*: $|S_1|=...=|S_N|=K$; $|S_k|$ desemnează dimensiunea lui S_k.

(c) *Regula Responsabilității Egale*: Fiecare proces j, 1<=j<=N este conținut în exact același număr D de seturi S_i, 1<=i<=N.

(d) S_i, 1<=i<=N, conține întotdeauna procesul i.

Proprietățile (b) și (c) sunt necesare pentru un algoritm complet distribuit; proprietate (d) este inclusă pentru a reduce cu 1 numărul de mesaje de trimise și de recepționate.

În condițiile de mai sus, se poate considera că un algoritm centralizat desemnează un singur proces ca arbitru (procesul coordonator). Acest arbitru este conținut în fiecare S_i și astfel este satisfăcută propricatea (b) cu k=1. Totuși, proprietatea (c) nu este satisfacută, deoarece nodul coordonator este singurul nod conținut în fiecare S_i.

4.3.2.2. Algoritmul lui Lamport

A. Descrierea algoritmului

În plus față de cerințele prezentate mai sus pentru realizarea excluderii reciproce, problema lui Lamport necesită ca cererile de acces la un obiect să fie deservite în ordinea în care au fost făcute. Algoritmul distribuit următor, cunoscut ca *algoritmul lui Lamport*, satisface toate cerințele problemei. Algoritmul impune următoarele:

(1) Mesajul in care se încapsulează o cerere de acces exclusiv de la procesul P_i conține o marcă de timp (TS_i, i) unde $TS_i=C_i$ este valoarea ceasului logic pentru procesul i, din momentul trimiterii mesajului.

(2) Fiecare proces menține o coadă de cereri de acces, inițial vidă, care conține mesajele ordonate după relația =>.

Algoritmul se definește prin cinci reguli; se presupune ce fiecare regulă se realizează printr-o operație indivizibilă:

(1) Cînd un proces cere accesul la obiectul partajat, el depune marca în coada de cereri proprie și apoi trimite un mesaj REQUEST (TS_i, i) la fiecare proces care participă la algoritm.

(2) Cînd un proces P_j recepționează mesajul REQUEST (TS_i, i) returnează imediat un mesaj REPLY însoțit de marca de timp și plasează mesajul recepționat în coada de mesaje proprie.

(3) Procesul P_i are acces la obiect cînd una din următoarele două condiții este satisfacută: (a) P_i are în varful cozii propria sa cerere; (b) P_i a primit un mesaj de la toate procesele cu o marcă de timp anterioară lui (TS_i, i).

BUPT

(4) Pentru a elibera obiectul, P_i înlatura mesajul REQUEST din coada sa proprie şi-i trimite un mesaj RELEASE, însoţit de o marcă de timp la toate procesele.

(5) Cînd un proces P_j recepţionează mesajul RELEASE de la P_i, înlatură din coadă mesajul REQUEST al acestuia.

Verificarea corectitudinii algoritmului se face pe baza observaţiilor următoare:

(1) Regula (3)(b) împreună cu presupunerea că mesajele sunt recepţionate în ordinea emisiei, garantează că P_i recunoaşte corect toate cererile care preced propria sa cerere.

(2) Deoarece relaţia => ordonează total cererile, regula (3)(a) garantează că se permite numai unui singur proces să aibă acces la un moment dat la un obiect.

Algoritmul satisface condiţiile (b) şi (c) pentru un algoritm distribuit (secţiunea 4.3.2.): fiecare proces menţine toate informaţiile şi simulează un proces planificator central.

Numărul total de mesaje necesare pentru a obţine şi elibera accesul exclusiv la obiect este 3*(N-1) : (N-1) mesaje REQUEST, (N-1) mesaje REPLY şi (N-1) mesaje RELEASE.

B.Implementare.

O descriere generală a implementării algoritmului lui Lamport se poate face utilizînd notaţia SR [And91].

Excluderea reciprocă este asigurată, de fapt, de un set de procese Server [1:*n*], cîte unul asociat fiecarui nod *i*. Cînd un proces *Server*[*i*], în urma unei cereri de intrare în secţiune critică primită de la clientul său asociat, decide că se poate intra în secţiune critică, deblochează clientul *i*, care intră astfel în secţiunea critică. La ieşirea din secţiunea critică, *Client*[*i*] semnalizează procesului *Server*[*i*].

```
type kind = enum (REQUEST, REPLY, RELEASE)           # tipuri de mesaje
chan remoteChan[1:n](kind, timeStamp: int, sender: int)   #canal pentru comunicatie in retea
chan localChan[1:n](kind)      # canal pentru transmiterea cererilor de intrare si de iesire din sectiunea
                               # critica (comunicatie locala intre clienti si server)
chan enterChan[1:n]( )         # canal pentru deblocarea unui client in asteptare la intrare in sectiune

Client [i: 1..n]::
    do true ->
            send localChan[i] (REQUEST)      # protocol de intrare in sectiunea critica
            receive enterChan[i] ( )
            #sectiune critica...
            send localChan[i] (RELEASE)      # protocol de iesire din sectiunea critica
    od

Server [i: 1..n]::
  var reqQueue: queue of (int,int)       # coada de cereri ordonata dupa marcile de timp
  var localTimeStamp: int                # marca de timp ce va fi asignata urmatoarei cereri locale
  var highTimeStamp: int := 0            # marca de timp cea mai mare dintre o marca de timp primita
                                         # intr-un mesaj si marca de timp locala
  var replyCounter: int                  # contorizeaza numarul de confirmari care trebuie asteptate
  var inAccess: bool := false            # indica accesul intr-o sectiune critica
  var knd: kind, sender: int, ts: int    # folosite la citirea mesajelor

  do true ->        # bucla infinita a severului
       if not (empty (localChan[i])) ->
            receive localChan[i](knd)
            if knd = REQUEST ->
                 localTimeStamp := highTimeStamp +1
                 replyCounter := n - 1
                 fa j := 1 to n ->
                        send remoteChan[j](REQUEST, localTimeStamp, i)
                 af
            [] knd = RELEASE ->
                 fa j := 1 to n st st j ≠ i ->
                        send remoteChan[j](RELEASE, localTimeStamp, i)
                 af
            fi
```


BUPT

```
        fi
        if not (empty(remoteChan[i])) →
                receive remoteChan[i](knd, ts, sender)
                if knd = REQUEST →
                        insereaza cererea in coada queue
                        highTimeStamp: = max(localTimeStamp, ts)
                        if sender ≠ i → send remoteChan[sender](REPLY, localTimeStamp, i) fi
                [] if knd = REPLY →
                        replyCounter := replyCounter - 1
                        if replyCounter = 0 and (cererea proprie este prima din coada)→
                                inAccesss := true;
                                send enterChan[i]( )
                        fi
                [] knd = RELEASE →
                        sterge din coada queue cererea de la sender
                        if replyCounter = 0 and (prima cerere din coada este o cerere proprie)→
                                inAccesss := true;
                                send enterChan[i]( )
                        fi
                fi
        fi
od
```

4.3.2.2. Algoritmul lui Ricard şi Agrawala

A. Descrierea algoritmului

Algoritmul lui Ricard şi Agrawala rezolvă aceeaşi problemă de excludere reciprocă, dar este mai eficient decat soluţia lui Lamport, deoarece necesită cel mult 2*(N-1) mesaje. Ca şi în algoritmul lui Lamport, cozile de cereri sunt total ordonate de relaţia =>. Algoritmul este descris prin următoarele reguli;

(1) Cînd un proces P_i cere accesul la obiectul partajat, trimite un mesaj REQUEST (TS_i, i) la următoarele procese.

(2) Cînd un proces P_j recepţionează mesajul REQUEST execută următoarele operaţii:

(a) Dacă P_j este într-o secţiune critică, reţine mesajul REQUEST al procesului i.

(b) Dacă P_j nu cere şi el acces la obiect, returnează un mesaj REPLY (însoţit de marca de timp).

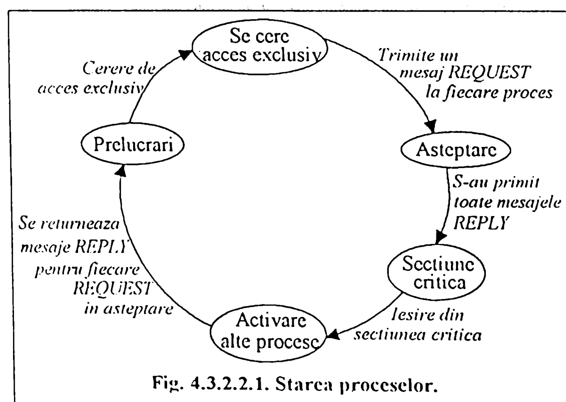


Fig. 4.3.2.2.1. Starea proceselor.

(c) Dacă procesul cere şi el acces la obiect şi marca de timp a cererii sale, (TS_j, j) precede pe (TS_i, i), atunci reţine mesajul REQUEST al procesului P_i; în caz contrar, returnează un mesaj REPLY.

(3) Cînd procesul P_i recepţionează mesajele REPLY de la toate nodurile, are acces exclusiv la obiect.

(4) Cînd procesul P_i eliberează obiectul, transmite un mesaj REPLY pentru fiecare mesaj REQUEST în aşteptare.

Diagrama de tranziţie a stărilor pentru fiecare proces este reprezentată în fig. 4.3.2.2.1.

BUPT

Analiza acestui algoritm relevă următoarele proprietăţi:

(1) Toate cererile vor fi total ordonate deoarece: (a) Unei cereri noi i se asignează un număr de secvenţă mai mare decât toate numerele de secvenţă ale oricăror cereri REQUEST cunoscute nodului; (b) Arbitrajul necesar în cazul unor cereri REQUEST trimise de noduri care nu au cunoştinţă fiecare de cererea celuilalt se face pe baza numerelor nodurilor.

(2) Controlul este complet distribuit.

(3) Interblocări nu pot apare deoarece nu pot apare cicluri cu noduri care aşteaptă la intrare în secţiunea critică.

(4) Amînarea nedefinită nu poate apare deoarece ordinea de deservire a cererilor este FIFO.

(5) Algoritmul este independent de caracteristicile comunicaţiei în sistem.

(6) Algoritmul necesită cel mult 2*(N-1) mesaje pentru a deservi o cerere.

Algoritmul are totuşi şi două dezavantaje:

(1) Fiecare proces trebuie să cunoască identitatea tuturor celorlalte procese din sistem. Cînd un proces nou se alatură unui grup de procese care participă la algoritmul de excludere reciprocă, sunt necesare următoarele acţiuni: (a) Procesul trebuie să primească numele tuturor proceselor din grup; (b) Numele noului proces trebuie făcut cunoscut tuturor proceselor din grup; Ambele acţiuni (a) şi (b) se fac avînd în vedere că pot exista mesaje REPLY şi REQUEST care circulă în sistem în momentul cînd noul proces se alătură grupului. Pot fi utilizate comunicaţii de grup sau fiecare proces menţine o listă cu toţi membrii grupului. Algoritmul funcţionează bine pentru grupuri mici, care nu îşi schimbă niciodată componenţa.

(2) Dacă unul dintre procese dispare, atunci întreaga schemă este compromisă. Această dificultate poate fi rezolvată prin monitorizare continuă a stării tuturor proceselor din sistem. Dacă un proces dispare, atunci restul proceselor sunt anunţate, pentru a nu mai trimite mesaje REQUEST acelui nod. În momentul readucerii lui în sistem, trebuie iniţializată procedura de alăturare a unui proces grupului.

B. Implementare

```
type kind = enum (REQUEST, REPLY, RELEASE)          # tipuri de mesaje
chan remoteChan[1:n](kind, timeStamp: int, sender: int)   #canal pentru comunicatie in retea
chan localChan[1:n](kind)        # canal pentru transmiterea cererilor de intrare si de iesire din sectiunea
                                 # critica (comunicatie locala intre clienti si server)
chan enterChan[1:n]( )           # canal pentru deblocarea unui client in asteptare la intrare in sectiune

Client [i: 1..n]::
    do true →
            send localChan[i] (REQUEST)      # protocol de intrare
            receive enterChan[i] ( )
            #sectiune critica...
            send localChan[i] (RELEASE)      # protocol de iesire
    od

Server [i: 1..n]::
  var localTimeStamp: int            # marca de timp ce va fi asignata urmatoarei cereri locale
  var highTimeStamp: int := 0        # marca de timp cea mai mare dintre o marca de timp primita
                                     # intr-un mesaj si marca de timp locala
  var replyCounter: int              # contorizeaza numarul de confirmari care trebuie asteptate
  var inAccess: bool := false        # indica accesul intr-o sectiune critica
  var reqAccess: bool := false       # indica o cerere de intrare in sectiune critica
  var later: bool := false           # indica necesitatea aminarii unui mesaj
  var laterTab[1:n]: bool := ([n] false)   # indica nodurile la care trebuie trimise confirmari la
                                           # iesirea din sectiunea critica
  var knd: kind, sender: int, ts: int      # folosite la citirea mesajelor

  do true ->        # bucla infinita a severului
        if not (empty (localChan[i])) →
                receive localChan[i](knd)
                if knd = REQUEST →
                        localTimeStamp := highTimeStamp + 1
                        reqAccess := true; replyCounter := n - 1
                        fa j := 1 to n st j ≠ i →
```


BUPT

```
                    send remoteChan[j](REQUEST, localTimeStamp, i)
                af
            [] knd = RELEASE →
                inAccess := false
                fa j := 1 to n st laterTab[j] = true →
                    send remoteChan[j](REPLY, localTimeStamp, i)
                    laterTab[j] := false
                af
            fi
        if not (empty(remoteChan[i])) →
            receive remoteChan[i](knd, ts, sender)
            if knd = REPLY →
                replyCounter := replyCounter - 1
                if replyCounter = 0 →
                    inAccesss := true; reqAccess := false
                    send enterChan[i]( )
                fi
            [] knd = REQUEST →
                highTimeStamp: = max(localTimeStamp, ts)
                later := (inAccess = true) or (reqAccess = true and ((ts>localTimeStamp)
                                  or (ts = localTimeStamp) and (sender > i)))
                if later = true → laterTab[sender] := true
                [] later ≠ true →send remoteChan[sender](REPLY,localTimeStamp, i)
                fi
            fi
        fi
od
```

4.3.2.3. Algoritm pentru excludere reciprocă în accesul la K resurse identice

A. Descrierea algoritmului

Algoritmii prezentati pînă acum permit numai unui singur proces să fie în secţiunea critică. Raymond [Ray87] a elaborat un algoritm, de fapt o extensie a algoritmului Ricard şi Agrawala, care permite intrarea a maxim K noduri în secţiunea critică. Aceasta problemă apare cînd există K obiecte identice care trebuie partajate între N procese.

Algoritmul se bazează pe observaţia că se permite unui proces să fie în secţiunea critică dacă nu mai mult de K-1 procese (din N-1) sunt, la acel moment, în secţiunea critică. Un proces care trimite N-1 mesaje pentru a anunţa dorinţa de a intra într-o secţiune critică, poate intra cînd primeste N-K mesaje REPLY, deoarece aceasta înseamnă că cel mult N-1-(N-K)=K-1 procese pot fi în secţiunea critică.

O diferenţă majoră între algoritmul lui Raymond şi algoritmul Ricard şi Agrawala constă în momentele în care pot apare mesaje REPLY, în cel de-al doilea, mesajele REPLY pot fi recepţionate numai cînd procesul aşteaptă la intrarea în secţiunea critică. În algoritmul Raymond, mesajele REPLY pot fi recepţionate de un nod cînd este în aşteptare la intrarea în secţiunea critică, cînd este în secţiunea critică sau după ieşirea din secţiunea critică. Pentru a distinge mesajele REPLY, trebuie menţinută de către fiecare proces o structură de date care să memoreze cîte mesaje REPLY mai sunt de recepţionat de la fiecare proces.

În algoritmul Ricard şi Agrawala, tabloul *laterTab[]* era de tip logic, deoarece un proces nu putea întîrzia decat cel mult un mesaj REPLY catre alt nod. În algoritmul Raymond este posibil de a amina mai multe mesaje REPLY către un nod; de aceea în tabloul *laterTab[]* elementele au valori întregi.

B. Implementare

```
const objectNo: int := k                          #numarul de obiecte disponibile
type kind = enum (REQUEST. REPLY. RELEASE)         # tipuri de mesaje
chan localChan[1:n](kind)        # canal pentru transmiterea cererilor de intrare si de iesire din sectiunea
                                 # critica (comunicatie locala intre clienti si server)
chan enterChan[1:n]( )           # canal pentru deblocarea unui client in asteptare la intrare in sectiune
chan remoteChan[1:n](kind, intValue: int, sender: int)    #canal pentru comunicatie in retea

Client [i: 1..n]::
```


BUPT

```
        do true →
                send localChan[i] (REQUEST)     # protocol de intrare
                receive enterChan[i] ( )
                #sectiune critica...
                send localChan[i] (RELEASE)     # protocol de iesire
        od
Server [i: 1..n]::
  var localTimeStamp: int         # marca de timp ce va fi asignata urmatoarei cereri locale
  var highTimeStamp: int := 0     # marca de timp cea mai mare dintre o marca de timp primita
                                  # intr-un mesaj  si marca de timp locala
  var replyCounter[1:n]: int  := ([n] 0)   # contorizeaza numarul de confirmari care trebuie
                                           # primite de la celelalte noduri
  var inAccess: bool := false      # indica accesul intr-o sectiune critica
  var reqAccess: bool := false     # indica o cerere de intrare in sectiune critica
  var later: bool := false         # indica necesitatea aminarii unui mesaj
  var laterTab[1:n]: int  :=  ([n] 0)# indica noduri le si numarul de confirmari care trebuie trimise la
                                   # iesirea din sectiunea critica
  var knd: kind, sender: int, intValRec; int  # folosite la citirea mesajelor

  do true ->        # bucla infinita a severului
        if not (empty (localChan[i])) →
                receive localChan[i](knd)
                if knd = REQUEST →
                        localTimeStamp := highTimeStamp +1
                        reqAccess := true;
                        fa j := 1 to n st j ≠ i →
                                send remoteChan[j](REQUEST, localTimeStamp, i)
                                replyCounter[i] := replyCounter[i] + 1
                        af
                [] knd = RELEASE  →
                        inAccess := false
                        fa j := 1 to n st laterTab[j] ≠ 0 →
                                send remoteChan[j](REPLY, laterTab[j], i)
                                laterTab[j] := 0
                        af
                fi
        fi
        if not (empty(remoteChan[i])) →
                receive remoteChan[i](knd, intValueRec, sender)
                        if knd = REPLY →
                                replyCounter [sender]:= replyCounter[sender]- intValRec
                                inregistreaza in cnt numarul de confirmari primite: este
                                numarul de elemente nule  din replyCounter
                                if (reqAccesss := true and  inAccess = false  and cnt ≥ n - objectNo) →
                                        inAccesss := true; reqAccess := false
                                        send enterChan[i]( )
                                fi
                        [] knd = REQUEST →
                                highTimeStamp: = max(localTimeStamp, ts)
                                later := (inAccess = true) or (reqAccess = true and ((ts > localTimeStamp)
                                                or (ts = localTimeStamp) and (sender > i)))
                                if later = true → laterTab[sender] :=  laterTab[sender] + 1
                                [] later ≠ true → send remoteChan[sender](REPLY, 1 , i)
                                fi
                        fi
        fi
  od
```


BUPT

4.3.2.4. Alți algoritmi

Alți algoritmi dezvoltați pentru rezolvarea problemei accesului exclusiv într-un sistem distribuit au avut drept obiectiv micșorarea și mai mult a numărului de mesaje. Astfel, Thomson [MOO87] a propus o *regula de consens* a majorității bazată pe ideea că, pentru a se obține acces la un obiect partajat este necesar permisiunea numai a majorității proceselor. Corect implementată, niciodată două procese nu pot acapara votul majorității deoarece intersecția a doua majorități este o mulțime nenulă de arbitri. De aceea cel puțin un arbitru va vota favorabil pentru o cerere și nefavorabil pentru cealaltă. Numărul de mesaje necesar pentru a obține permisiunea de acces la un obiect partajat este, în cel mai bun caz, $n/2$. Extensii ale acestei metode, propuse de Gifford [MOO87] și Skeen permit unor procese să contribuie cu mai mult de un vot. În aceste scheme este suficient să se obțină o majoritate a voturilor, care nu este neapărat o majoritate a arbitrilor. Printr-o corectă distribuție a voturilor, algoritmul poate varia de la o formă centralizată la o forma distribuită. De exemplu, dacă unui proces i se acordă toate voturile, în timp ce celorlalte nu li se acordă nici un vot, algoritmul este redus la un control central.

Este evident că algoritmii Lamport, Ricard și Agrawala și Thomas satisfac proprietățile (a), (b) și (c) prezentate în secțiunea 4.3.2, cu $K=D=n$. În general, algoritmii cu voturi de greutate diferită, nu satisfac proprietatea (c). Dacă greutatea voturilor nu este egal distribuită printre procese, atunci algoritmul rezultat nu este total distribuit. Un algoritm care reduce și mai mult numărul de mesaje necesare pentru obținerea accesului exclusiv este algoritmul lui Maekawa[MOO87]; acesta alege pentru K și D valorile minime care satisfac condițiile (b) și (c).

4.3.2.5. Consideratii finale

În secțiunea 4.3.2 s-a considerat că sistemul distribuit se bazează pe o rețea complet conectată ceea ce înseamnă că există o linie de comunicație directă între oricare două noduri. Dacă această cerință nu este satisfacută, la optimalitatea unui algoritm pentru excludere mutuală contribuie și un alt factor, numărul total de noduri intermediare necesare pentru a trimite un mesaj de la sursa la destinație. Pe de altă parte, unele caracteristici speciale ale rețelei pot permite și alte optimizari ale algoritmilor.

A. Topologii speciale ale rețelei

Dacă topologia rețelei permite transmiterea de mesaje broadcast, atunci, pentru a transmite mesajele REQUEST un proces poate transmite doar un singur mesaj și nu (n-1).

Dacă topologia rețelei este un inel, algoritmul poate fi modificat pentru a reduce numărul de mesaje și pentru a minimiza numărul de noduri intermediare. Următoarele modificari ale algoritmului elimină necesitatea mesajelor REPLY:

(1) Nodul care cere acces la obiect trimite un mesaj REQUEST la vecinul sau din inel. Mesajele traversează într-o direcție inelul.

(2) Nodul vecin recepționează mesajul REQUEST; dacă acesta are prioritate adecvată, este trimis la vecinul următor.

(3) Cînd mesajul REQUEST este returnat la nodul inițiator, se permite accesul la obiect.

Astfel, sunt permise n noduri intermediare. Algoritmul este optimal relativ la numărul de noduri intermediare și numărul de mesaje.

B. Modificari dinamice ale topologiei rețelei

Un nod care se alătură sistemului distribuit, trebuie să execute următorii pași:

(1) Să interogheze nodurile active pentru a obține o listă a tuturor nodurilor participante.

(2) Să-și asigneze un număr de nod propriu, unic.

(3) Să-și plaseze numărul de nod pe listele tuturor nodurilor participante.

(4) Să-și stabilească valoarea adecvată pentru marca de timp inițiala.

Pasul (2) poate fi complicat dacă se permite o asignare dinamică a numerelor de noduri. O soluție simplă este de a asigna numere fixe și unice fiecarui nod, indiferent dacă este activ sau nu.

În pasul (4), marca de timp trebuie să fie mai mare decat marca oricărui mesaj REQUEST care ar fi putut fi primit dacă nodul ar fi fost activ continuu; aceasta implică comunicare cu celelalte noduri. Pînă la satisfacerea acestei condiții, noul nod nu poate cere acces la obiect și trebuie să întîrzie orice mesaj REPLY necesar pentru un REQUEST primit.

Un nod care părăsește voluntar sistemul, trebuie să-și anunțe intenția celorlalte noduri. Numai după primirea unor confirmari de la toate nodurile, îi este permis să părăsească rețeaua; aceasta înseamnă că, în timp ce așteaptă confirmările, nu poate participa la comunicații într-un mod care să afecteze excluderea reciprocă (de exemplu, nu mai poate cere acces la obiect și trebuie să returneze REPLY la orice mesaj REQUEST primit).

Pentru a detecta o defecțiune a unui nod, se poate folosi un mecanism de timeout. La trimiterea unui mesaj REQUEST, nodul emițător startează un timer. Dacă timerul detectează timeout, se trimite un mesaj de probă, timp în care se consideră că nodul respectiv nu mai este în sistem. Dacă nodul nu răspunde nici la mesajul de probă, se inițiază procedura de înlaturare a unui nod.

BUPT

C. Gestiunea numerelor de secventa ale mesajelor

Dacă traficul de mesaje este mare şi toate nodurile sunt participante la decizia de acces la obiect, ceasurile logice vor fi sincronizate foarte strîns. Ca rezultat, regula de prioritate a proceselor va tinde să favorizeze nodurile cu numere mici. Acest lucru poate fi evitat prin utilizarea unei creşteri aleatoare a numărului de secvenţă; şi mai bine, se poate introduce deliberat o nouă prioritate dacă se permite nodurilor cu prioritate mare să utilizeze creşteri mici ale numerelor de secvenţă, iar nodurile de prioritate mică să utilizeze creşteri mari.

4.3.2.6. Implementarea algoritmilor distribuiţi pentru excludere reciprocă în reţele Unix, MS Windows (Win32) şi Novell NetWare

În această secţiune se prezintă cîteva soluţii de implementare a algoritmilor Lamport, Ricard & Agrawala, şi Raymond în reţele locale Novell NetWare 3.11, Unix şi Microsoft Windows (utilizînd funcţii Win32).

A. Implementarea în reţele Novell NetWare.

Implementarea unui program demonstrativ care înglobează atit serverul cit şi clientul din descrierea algoritmilor SR este prezentată în anexa B. Se foloseşte pentru comunicaţie protocolul IPX şi facilitatea de a se putea primi şi trata mesaje asincron faţă de procesul în curs de execuţie, utilizînd rutine de tratare a întreruperilor declarate Event Service Routine (ESR). Acestea vor recepţiona mesajele REQUEST, REPLY şi RELEASE.

Implementarea pune la dispoziţia programelor utilizator, într-o bibliotecă, următoarele rutine, pentru a căror utilizare trebuie inclus header-ul "exc.h":

(1) **int ExclAccessInitialize(void)** pentru iniţializarea blocului de control al operaţiilor de recepţie a mesajelor, deschiderea soclurilor pentru emisie şi recepţie, instalarea rutinei ESR şi intrarea în ascultare de pachete pentru recepţie. Funcţia returnează un cod de eroare adecvat dacă iniţializările nu pot fi efectuate. Eroarea IPX care provoacă eşuarea iniţializarilor este indicată în variabila externa *code*.

(2) **void ExclAccessOpen(void)** pentru obţinerea accesului într-o secţiune critică, conform algoritmului. Rutina apelează două funcţii ale sistemului, Request() pentru transmiterea în reţea a cererii de intrare în secţiune şi Enter() care testeză în buclă continuă valoarea unei variabile setată de rutina ESR atunci cind sunt îndeplinite toate condiţiile de intrare în secţiune; pentru algoritmul Ricard & Agrawala condiţia de intrare în secţiunea critică este ca toate nodurile să fi trimis mesaje REPLY (*replyMesNo*=0), pentru algoritmul Raymond numărul de noduri care au trimis mesaje REPLY trebuie să fie mai mare sau egal cu numărul de noduri minus numărul de obiecte disponibile iar pentru algoritmul Lamport condiţia este ca în lista de cereri de intrare, ordonată în raport de mărcile de timp, cererea proprie să fie prima şi să fi fost confirmată de toate celelalte noduri.

(3) **void ExclAccessClose(void)** pentru ieşirea dintr-o secţiune critică, cînd trebuie trimise mesaje REPLY (în cazul algoritmilor Ricard & Agrawala şi Raymond) sau mesaje RELEASE (în cazul algoritmului Lamport). Aceste mesaje pot provoca deblocarea altor clienţi care aşteaptă la intrarea într-o secţiune critică.

(4) **void ExclAccessEnd(void)** pentru deconectarea apelantului din sistem: închiderea soclurilor şi întreruperea ascultării de pachete pentru recepţie.

(5) **int SetUsersAndObjects (char *users[MAX_CON][USER_LEN]*, int *hosts*, int *noObjects*)** pentru iniţializarea structurilor de date ce trebuie menţinute pentru fiecare nod: numărul conexiunii cu serverul Novell şi adresa Ethernet. Funcţia primeşte ca parametri de intrare numele de user sub care s-a facut conectarea în fiecare nod care participă la algoritm, numărul de noduri şi, pentru algoritmul Raymond, numărul de obiecte disponible (pentru restul algoritmilor, acest parametru se transmite întotdeauna 1) şi returnează un cod de eroare care reprezintă numărul primului user din lista *users* care nu este conectat în reţea şi al cărui număr de conexiune, evident, nu poate fi determinat.

Pentru demostrarea corectitudinii tuturor algoritmilor pentru excludere reciprocă a fost scris un singur program utilizator general care permite alegerea algoritmului (anexa B).

B. Implementare Unix şi Microsoft Windows

Sistemele de operare Unix şi Microsoft Windows (Windows NT, Windows 95) permit o implementare mai apropiată de descrierea SR a algoritmilor prezentată în secţiunile 4.3.2.1., 4.3.2.2. şi 4.3.2.3. Protocolul utilizat, avind în vedere că se lucrează în reţele locale, este UDP.

Unul dintre scopurile implementării a fost ca ea să fie valabilă, cu modificări minime, atît în sisteme Unix cît şi Microsoft Windows (deci în reţele mixte). Pentru toţi algoritmii de excludere reciprocă a fost realizat un singur program client demonstrativ (aplicaţie XWindws, respectiv Windows) a cărui funcţie este de a transmite cererile de intrare şi ieşire din secţiunea critică ale utilizatorului, prin intermediul unui soclu UDP local (canalul *localChan* din descrierile algoritmilor) la procesul server. Serverul şi clientul din descrierea algoritmilor sunt implementate ca procese separate.

BUPT

Procesul server constă într-o buclă infinită în care aşteaptă recepţionarea de mesaje de la clientul local, pe soclul UDP local, sau de la un proces server de pe alt nod, pe soclul *remoteChan*. Pentru a detecta cînd s-au primit mesaje pe aceste socluri se foloseşte funcţia *select()*. Pentru deblocarea unui client în aşteptare la intrare într-o secţiune critică se foloseşte, în Windows un semafor iar în Unix, un canal *pipe*; acesta constituie de fapt implementarea canalului *enterChan*.

La dispoziţia unui program client se pun următoarele funcţii (anexa B):
(1) int ExclAccessInitialize(void) pentru deschiderea semaforului sau *pipe*-ului folosit, determinarea adresei proprii şi deschiderea soclului UDP local. Funcţia returnează un cod de eroare în cazul că nu se pot face iniţializările.
(2) void ExcAccessOpen(void) pentru obţinerea accesului în secţiunea critică. Funcţia transmite pe soclul local mesajul "REQUEST" serverului şi blochează clientul, în urma unui apel Win32, WaitForSingleObject() la semaforul folosit pentru aşteptare la intrarea în secţiunea critică; deblocarea clientului se face de catre procesul server, printr-un apel Win32, ReleaseSemaphore() cînd au fost îndeplinite toate condiţiile pentru intrare.
(3) char* GetMyAddress(void) pentru obţinerea adresei IP proprii. Funcţia returnează adresa unui şir la care a depus adresa IP a propriului nod.
(4) void ExclAccessClose(void) apelată la ieşirea dintr-o secţiune critică.

Pentru procesele server nu a fost prevazută o interfaţă grafica; ele se execută, în mod normal, în background. Pentru determinarea adreselor nodurilor care participă la algoritm s-a adoptat următoarea soluţie: serverele se lansează în execuţie avînd ca parametri un identificator numeric al serverului şi numărul total de noduri care participă la algoritm. La începerea execuţiei, fiecare server transmite broadcast identificatorul său şi adresa sa IP proprie şi aşteaptă recepţionarea adreselor IP ale tuturor nodurilor pe care le memorează într-o tabelă, după care intră în bucla infinită de aşteptare şi tratare a mesajelor.

Există şi alte soluţii de implementare a serverelor; s-a preferat însă soluţia care utilizează *select()* pe de o parte, datorită consideraţiilor prezentate la 3.1.2. şi, pe de altă parte, datorită universalităţii ei. Soluţia este valabilă, aproape identic, atît în reţele Unix cît şi în reţele Windows NT şi Windows 95; singura deosebire este faptul că, în Win32 nu există funcţia *getdtablesize()* obligatorie însă în Unix înainte de utilizarea lui *select()*. O altă soluţie ar putea fi aceea în care prelucrarea mesajelor locale şi de la distanţă se face în două fire separate; un fir se deblochează la recepţia unui mesaj pe soclul *localChan* iar celălalt la recepţia unui mesaj pe soclul *remoteChan*.

4.3.3. Algoritmi care utilizează un token circulant pentru excludere reciprocă

O altă metodă de a obţine excluderea reciprocă constă în circulaţia unui token între procesele sistemului. Un token este un mesaj de un tip special care se transmite în întregul sistem. Posesia tokenului permite procesului respectiv să intre în secţiunea critică. Deoarece există un singur token în sistem, numai un singur proces se poate afla la un moment dat în secţiunea critică.

4.3.3.1. Sisteme structurate în inel

A. Descrierea algoritmului

Presupunem că procesele din sistem sunt organizate logic într-o structură sub forma de inel; aceasta nu înseamnă că topologia reţelei trebuie să fie neaparat inel. Se poate utiliza, de exemplu o reţea bus (Ethernet) fără nici o ordine predefinită a proceselor şi, se poate stabili, prin software, o ordine oarecare a lor, de exemplu în ordine crescătoare a adreselor fizice de reţea a nodurilor lor; ceea ce contează este ca fiecare proces să cunoască adresa vecinului sau. Cînd se construieşte inelul, procesului 0 i se repartizează tokenul. Tokenul este transmis de la procesul i la procesul i+1. Cînd un proces primeste tokenul fie (1) dacă vrea să intre în secţiunea critică, pastrează tokenul pînă la ieşirea din secţiunea critică cînd îl transmite nodului următor (vecinului din inel), fie (2) dacă nu vrea să intre în secţiunea critică, transmite imediat tokenul vecinului.

Excluderea reciprocă este asigurată, de fapt, de un set de procese *Server*[1:*n*], cîte unul asociat fiecărui proces *Client*[1:]; procesele *Server* formează inelul şi partajează tokenul. Cînd un proces *Server*[i] posedă tokenul, deblochează procesul său asociat *Client*[i], care intră astfel în secţiunea critică. La ieşirea din secţiunea critică, *Client*[i] semnalizează procesului *Server*[i] care transmite tokenul mai departe, procesului *Server*[i mod *n*+1].

Deoarece inelul este unidirecţional, amînarea nedefinită a unui proces nu poate avea loc. Numărul de mesaje necesare pentru implementarea excluderii reciproce variază de la un mesaj pe intrare, în cazul unei concurenţe mari (fiecare proces doreşte intrarea în secţiunea critică) la infinit, în cazul în care nici un proces nu doreşte intrarea în secţiunea critică.

Există două probleme legate de acest algoritm: (1) dacă tokenul este pierdut, el trebuie regenerat. Detectarea pierderii tokenului nu este o operaţie simplă, deoarece intervalul de timp dintre două apariţii succesive ale tokenului nu este limitat. Ca atare, cînd tokenul este pierdut este necesar un algoritm de alegere

BUPT

iar procesul ales să genereze tokenul. (2) dacă un proces (*server*) dispare, trebuie restabilit un nou inel logic. Există mai mulţi algoritmi de alegere şi de reconstrucţie a unui inel logic ([Lan77] [Lan83][GMO82] (vezi subcapitolul 4.4); în continuare va fi prezentat algoritmul lui Lelann).

Algoritmul lui Lelann permite detecţia unui token pierdut şi regenerarea lui. Se presupune că fiecare proces are un timer. Acest timer este resetat la recepţia ***tokenului de control*** (CT) sau alt token; tokenurile circulă în sistem după regula FIFO. Protocolul propus de Lelann este un alt exemplu de algoritm distribuit; principala structură de date utilizată de algoritm este o listă cu numerele de priorităţi ale tuturor proceselor active, denumită ***lista activă*** (fiecare proces menţine propria sa lista activă). Protocolul se descrie astfel:

(1) Cînd timpul setat de timer s-a epuizat, procesul corespunzator P_i generează un ***token de alegere*** (ET(i)) care conţine numărul procesului. Timerul este setat din nou; se spune că s-a intrat în ***faza de alegere***. Totodată în lista activă proprie, S(i), se înscrie i.

(2) Cînd un candidat primeste înapoi tokenul CT înainte de propriul sau token ET, anulează faza de alegere şi înlatură tokenul ET cînd îl primeşte.

(3) Cînd un candidat P_i primeste de la vecinul din stanga un token ET(j) atunci: (1) Dacă acesta este primul token de alegere primit sau trimis, P_i crează o listă activă nouă, cu numerele i şi j şi trimite apoi ET(i) urmat de ET(j); (2) Dacă i<>j, atunci P_i adauga j la lista activă şi trimite mesajul mai departe la vecinul din dreapta.

(4) Cînd un candidat P_i primeste propriul sau ET(i), îl înlatură din inel, resetează timerul şi execută următorul algoritm: Dacă i=max S(i) atunci se generează un nou token CT şi se setează timerul.

Eficienţa mecanismului de sincronizare cu token circulant depinde mult de dimensiunea (în timp) a secţiunii critice. Este evident că, atunci cînd secţiuniile critice implică schimburi de mesaje între producatori şi consumatori acest mecanism de sincronizare nu oferă performanţe bune. În schimb, acest mecanism a fost folosit pentru dezvoltarea unui algoritm pentru detectarea terminării unei prelucrari distribuite ([And91] [DS83] [Mis83]); circulaţia tokenului se foloseşte pentru a semnaliza modificările de stare.

B. Implementare

```
type kind = enum (REQUEST, RELEASE)   # tipuri de mesaje
chan remoteChan[1:n]( )              #canal pentru comunicatie in retea
chan localChan[1:n](kind)            # canal pentru transmiterea cererilor de intrare si de iesire din sectiunea
                                     # critica (comunicatie locala intre clienti si server)
chan enterChan[1:n]( )               # canal pentru deblocarea unui client in asteptare la intrare in sectiune

Client [i: 1..n]::
        do true →
                send localChan[i] (REQUEST)      # protocol de intrare
                receive enterChan[i] ( )
                #sectiune critica...
                send localChan[i] (RELEASE)      # protocol de iesire
        od

Server [i: 1..n]::
  var doAccess: bool := false       # indica accesul intr-o sectiune critica
  var knd: kind                     # folosita la citirea mesajelor locale
  do true ->        # bucla infinita a severului
        if not (empty (localChan[i])) →
                receive localChan[i](knd)
                if knd = REQUEST → doAccess := true;
                [] knd = RELEASE →
                        doAccess := false
                        send remoteChan[i mod n + 1]( )
                fi
        fi
        if not (empty(remoteChan[i])) →
                receive remoteChan[i]( )
                        if doAccess = true → send enterChan[i]( )
                        [] doAccess ≠ true → send remoteChan[i mod n + 1]( )
                        fi
        fi
  od
```


BUPT

4.3.3.2. Algoritmul pentru sisteme nestructurate în inel

În contrast cu sistemele structurate în inel, în cele nestructurate în inel un proces care deține tokenu poate să-l trimită oricarui alt proces din sistem. În plus, nu este nevoie ca tokenul să circule dacă nici ur proces nu cere să intre în secțiunea critică.

Algoritmul următor aparține lui Chandy ([Gos91] [Cha82] [SP88]) și garantează că fiecare proce: care cere intrarea într-o secțiune critică va primi tokenul într-un interval de timp finit față de momentu lansării cererii.

Se consideră că există *n* procese în sistem. Tokenului i se asociază un vector $T=(T_1,T_2,...T_n)$, unde T indică numărul de intrări în secțiunea critică a procesului P_i. Algoritmul constă în următoarele etape:

(1) Cînd un proces P_i cere să intre în secțiunea critică pentru a m_i-a oară, trimite mesajul REQUEST(P_i,m_i catre toate celelalte procese și așteaptă apoi să primească tokenul.

(2) Cînd procesul P_i primește tokenul, setează $T_i=m_i$ în vectorul T al tokenului și intră în secțiunea critică.

(3) După ce procesul P_i iese din secțiunea critică, examinează coada de cereri de acces primite:

- (a) Dacă coada este vidă, procesul P_i continuă execuția normală (pînă primește mesaje de la alt procese)
- (b) Dacă coada nu este vidă, procesul P_i preia prima cerere din coadă, fie aceasta (P_j, m_j) Coada est ordonată FIFO.
- (c) Dacă $m_j>T_j$ atunci P_i trimite tokenul procesului P_j; în caz contrar, elimină cererea, ca fiind o cerer veche, deja deservită și preia din coadă următoarea cerere, pe care o prelucrează ca mai sus.
- (d) Aceasta prelucrare continuă pînă cînd coada este vidă sau tokenul este trimis altui proces.

```
type kind  =  enum (REQUEST, TOKEN, RELEASE)          # tipuri de mesaje
chan  localChan[1:n](kind)          # canal pentru transmiterea cererilor de intrare si de iesire din sectiunea
                                    # critica (comunicatie locala intre clienti si server)
chan   enterChan[1:n]( )            # canal pentru deblocarea unui client in asteptare la intrare in sectiune
chan  remoteChan[1:n](kind, sender: int,  case kind of
                                              REQUEST: (accesses: int)
                                              TOKEN:    (T[1..n]: int)
                          )  #canal pentru comunicatie in retea
Client [i: 1..n]::
        do true →
              send localChan[i] (REQUEST)      # protocol de intrare
              receive enterChan[i] ( )
              #sectiune critica...
              send localChan[i] (RELEASE)      # protocol de iesire
        od

Server [i: 1..n]::
  var   inAccess: bool := false        # indica accesul intr-o sectiune critica
  var   tokenOwner: bool := false      # indica detinerea tokenului
  var   reqQueue[1:n]: int :=  ([n] 0)# tablou care inregistreza cererile proceselor
  var   T[1:n]: int := ([n] 0)         # token transmis in  sistem
  var   knd: kind, sender: int         # folosite la citirea mesajelor
  var   accesses: int := 0             # folosit pentru a contoriza cererile pentru sectiune critica

  do true ->         # bucla infinita a severului
        if not (empty (localChan[i])) →
              receive localChan[i](knd)
              if knd = REQUEST →
                    accessses := accesses + 1
                    reqAccess := true;
                    fa j := 1 to n  →
                          send remoteChan[j](REQUEST,  i, accesses)
                    af
              || knd = RELEASE  →
                    inAccess := false
                    reqQueue[i] :=  0
```


BUPT

```
                fa j := 1 to n st reqQueue[j] ≠ 0  and (tokenOwner = true) →
                        if reqQueue[j] > T[j] →
                                send remoteChan[j](TOKEN, i, T)
                                tokenOwner := false
                        [] reqQueue[j] := 0
                        if
                af
        fi
fi
 if not (empty(remoteChan[i])) →
        receive remoteChan[i](knd, sender)
        if knd = REQUEST →
                receive remoteChan[i](reqQueue[sender])
                if (tokenOwner := true and  inAccess = false) →
                        if (reqQueue[i] ≠ 0 →
                                inAccesss := true; T[i] := reqQueue[i]
                                send enterChan[i]( )
                        [] reqQueue[i] = 0 →
                                fa j := 1 to n st reqQueue[j] ≠ 0  and (tokenOwner = true) →
                                        send remoteChan[j](TOKEN,  i, T)
                                tokenOwner := false
                                af
                        fi
        [] knd = TOKEN →
                receive remoteChan[i](T)
                T[i] := reqQueue[i]
                inAccesss := true; tokenOwner := true
        fi
fi
od
```

4.3.3.3. Secvențiator circulant

A. Descrierea algoritmului

O altă metodă bazată pe circulația dreptului de acces folosește un secvențiator circulant ([Lan83] [Gos91]). Un *secvențiator circulant* este un secvențiator care circulă permanent în inelul logic al proceselor, via unui mecanism de token circulant (token de control).

Modalitatea de a obține excluderea reciprocă de procesele care partajează secvențiatorul este următoarea: (1) La primirea tokenului, un proces poate executa cîteva primitive *ticket*; (2) Apoi trimite tokenul la succesorul său în inel.

Dacă se consideră *p* procese și *n* acțiuni în așteptare pînă ce procesul *p* primește tokenul, strategia constă în a executa exact *n* consecutive operații ticket deci a obține *n* tickete consecutive. În unele cazuri însă, necesitatea de a aștepta tokenul pentru a executa o acțiune poate fi considerată inacceptabilă. Pentru aceste cazuri, Lelann prezintă diferiți algoritmi care permit selecția anticipată a ticketelor. În funcție de strategia de selecție a ticketelor, procesele consumatoare ale ticketelor pot utiliza relația de ordine stabilită prin numerele ticketelor fie pentru a evita planificări inconsistente, fie pentru a detecta apariția unor planificări inconsistente și a lua decizia necesară. De exemplu, conflictele între acțiunile corespunzătoare a două sau mai multor tranzacții asupra unor copii ale aceluiași obiect (fișiere replicate, baze de date partiționate) pot fi rezolvate în faza de commit a tranzacțiilor, deci după ce acțiunilor li s-a permis execuția, pe baza priorităților date de numerele ticketelor.

B. Implementare

```
type kind  =  enum (REQUEST. TICKET, RELEASE)        # tipuri de mesaje
chan  remoteChan[1:n]( seqValue: int)                 #canal pentru comunicatie in retea
chan  localChan[1:n](kind)          # canal pentru transmiterea cererilor
chan  enterChan[1:n]() # canal pentru deblocarea unui client in asteptare la intrare in sectiune
chan  answerChan[1:n](seqValue: int) # canal pentru obtinerea unui ticket
Client [i: 1..n]::
```


BUPT

```
var ticket: int = 0            #valoarea ticketului primita de la server
    do true →
        send localChan[i] (REQUEST)     # protocol de intrare
        receive enterChan[i] ( )
        #sectiune critica...
        send localChan[i] (TICKET)      # protocol de obtinere a ticketului
        receive answer[i] (ticket)
        send localChan[i] (RELEASE)     # protocol de iesire
    od


Server [i: 1..n]::
 var  doAccess: bool := false      # indica accesul intr-o sectiune critica
 var  knd: kind                    # folosita la citirea mesajelor locale
 var ticketValue: int = 0          # valoarea curenta a ticketului

 do true ->                   # bucla infinita a sevrului
    if not (empty (localChan[i])) →
        receive localChan[i](knd)
        if knd = REQUEST → doAccess := true;
        || if knd = TICKET → send answerChan[i](ticketValue )
        || knd = RELEASE →
                doAccess := false
                send remoteChan[i mod n + 1](ticketValue )
        fi
    fi
    if not (empty(remoteChan[i])) →
        receive remoteChan[i](ticketValue)
                if doAccess = true → send enterChan[i]( )
                || doAccess ≠ true → send remoteChan[i mod n + 1]( )
                fi
    fi
 od
```

4.3.3.4. Implementarea algoritmilor care utilizează un token circulant în rețele Unix, MS Windows (Win32) și Novell NetWare

A. Implementare Novell NetWare.

Se utilizează același program demonstrativ prezentat la 4.3.2.6. Analog, implementarea pune la dispoziția programelor client următoarele rutine (anexa B):

(1) **int ExclAccessInitialize(void)** care are aceleași funcții ca și în 4.3.2.6., adică deschiderea soclurilor pentru emisie și recepție, inițializarea blocului de control pentru recepția mesajelor, instalarea rutinei ESR, intrarea în ascultare de pachete pentru recepție, dar, în plus, are și funcția de a genera token-ul în nodul cu identificatorul cel mai mic. Funcția returnează un cod de eroare sau zero în cazul execuției cu succes.

(2) **void ExclAccessOpen(void)** cu rolul de a obține accesul în secțiunea critică, atunci cînd este primit token-ul.

(3) **void ExclAccessClose(void)** apelată la ieșirea dintr-o secțiune critică, pentru transmiterea token-ul.

(4) **void ExclAccessEnd(void)** apelată pentru deconectarea apelantului din sistem: închiderea soclurilor și întreruperea ascultării de pachete pentru recepție.

(5) **int SetUsersAndObjects (char *users[MAX_CON][USER_LEN]*, int *hosts*, int *noObjects*)** pentru inițializarea structurilor de date ce trebuie menținute pentru fiecare nod: numărul conexiunii cu serverul Novell și adresa Ethernet. Tabela numerelor de conexiune a nodurilor care participă la algoritm este ordonată crescător. În cazul algoritmului în inel, această tabelă se utilizează pentru a determina adresa nodului vecin, iar în cazul algoritmului Chandy, această tabelă se utilizează pentru a trimite token-ul la nodul cu numărul de conexiune cel mai mic dintre toate nodurile care așteaptă la intrarea într-o secțiune critică. În cazul algoritmului Chandy se asociază astfel o prioritate fiecarui nod.

BUPT

B. Implementare Unix şi Windows.

În cazul reţelelor Unix şi Windows se utilizează acelaşi program client demonstrativ prezentat la 4.3.2.6. Spre deosebire de varianta Novell, clientul şi serverul se înglobează însă în procese diferite. Canalul *enterChan* se implementază în Windows sub forma unui semafor iniţializat la zero; după lansarea unei cereri de intrare în secţiunea critică, un client aşteaptă blocat la acest semafor. În Unix, blocarea clienţilor se realizează prin intermediul unui *pipe*. În cazul secvenţiatorului circulant, clientul trebuie să primească valoarea lui. În Windows, comunicarea server-client s-a realizat printr-o zonă de memorie partajată iar în Unix printr-un *pipe*. Implementarea pune la dispoziţia programelor client următoarele funcţii (anexa B):

La dispoziţia unui program client se pun următoarele funcţii:

(1) int ExclAccessInitialize(void) pentru iniţializari: deschiderea semaforului sau *pipe*-ului pentru sincronizarea client-server, determinarea adresei proprii şi deschiderea soclului UDP local.

(2) void ExcAccessOpen(void) pentru obţinerea accesului la secţiunea critică.

(3) char* GetMyAddress(void) pentru obţinerea adresei IP proprii. Funcţia returnează adresa unui şir la care a depus adresa IP a propriului nod.

(4) void ExclAccessClose(void) apelată la ieşirea dintr-o secţiune critică.

Tabela adreselor IP ale nodurilor se ordonează în raport de valoarea identificatorilor numerici, pentru a se putea determina adresa nodului vecin, în cazul algoritmului în inel sau pentru a se obţine priorităţile alocate nodurilor în cazul algoritmului Chandy.

4.3.4. Algoritmi broadcast

În unele tipuri de reţele locale, nodurile partajează un canal de comunicaţie comun. În aceste cazuri, se poate considera că fiecare nod este conectat direct cu celelalte. Deseori, în aceste tipuri de reţele, suportul software oferă o primitivă specială, broadcast care transmite un mesaj de la un nod la toate celelalte.

Fie *P*[1..*n*] *n* procese şi channel[1..*n*] *n* canale de comunicaţie, cîte unul pentru fiecare proces. Execuţia unei primitive broadcast channel(*mesaj*) are ca efect plasarea cîte unei copii a mesajului pe fiecare canal channel[1..*n*], asociat procesului *P*[*i*]; acest efect este echivalent cu execuţia în paralel a *n* primitive *send*, fiecare trimiţînd un mesaj pe unul dintre cele *n* canale. Procesul *i* recepţionează mesajul transmis, în urma execuţiei primitive receive channel[*i*].

Primitiva *broadcast* poate fi exploatată în proiectarea unor algoritmi de sincronizare a proceselor în sisteme distribuite, toţi bazîndu-se pe utilizarea primitivei *broadcast*. Problema centrală a acestor algoritmi o constituie realizarea ordonării totale a evenimentelor. Aceasta va fi rezolvată folosind ceasurile logice astfel: (1) fie LC valoarea timpului logic curent în procesul A; cînd A trimite un mesaj (nu neaparat broadcast) se înglobează în mesaj şi LC-ul curent, după care se incrementează LC (LC:=LC+1).; (2) cînd A recepţionează un mesaj cu marca de timp TS de la un proces B, se setează timpul logic curent LC la max(LC, TS) şi, apoi se incrementează LC.

Algoritmii broadcast sunt adecvaţi pentru sincronizarea unui număr mic de procese, avînd în vedere că toate procesele trebuie să proceseze fiecare mesaj.

4.3.4.1. Semafoare distribuite

Semafoarele se implementează, în mod normal, în sistemele centralizate, fie utilizînd variabile partajate, fie, în sistemele bazate pe transmisia de mesaje, utilizindu-se un proces server (cu rol de monitor). În subcapitolul de faţă, se prezinta o implementare a semafoarelor în sisteme distribuite la baza căreia stă utilizarea promitivei broadcast [And91].

Un senamfor s este un tip de date abstracte, care poate fi accesat numai prin intermediul a două operaţii primitive: Down(s) (sau P(s)) şi Up(s) (sau V(s)). Aceste operaţii se sincronizează astfel încît întotdeauna se menţine un *invariant* pentru semafoare: valoarea dintre numărul de operaţii Down complet executate este cel mult numărul de operaţii Up complet executate plus valoarea iniţială a semaforului.

Intr-o implementare a semafoarelor utilizînd variabile partajate, s se reprezintă printr-o variabila de tip întreg, nenegativa. Execuţia unei operaţii Up(s) incrementează aceasta variabila printr-o acţiune atomică; execuţia unei operaţii Down(s) este întîrziată pînă cînd variabila s are o valoare pozitivă, după care se decrementează, tot printr-o acţiune atomică. Într-un sistem distribuit, este însă necesară o altă tehnică pentru a reprezenta valoarea semaforului şi pentru a menţine invariantul pentru semafoare. În principal este necesară contorizarea operaţiilor Down(s) şi Up(s) şi întîrzierea execuţiei operaţiilor Down(); în plus, procesele care partajează un semafor trebuie să coopereze astfel încît să păstreze invarianţa semaforului chiar dacă starea programului este distribuită.

Cerinţele de mai sus pot fi realizate dacă se impune ca, înainte de execuţia unei operaţii Down sau Up, fiecare proces să transmită această intenţie printr-un mesaj broadcast şi să exemineze răspunsurile primite pentru a determina dacă execuţia poate continua sau nu. Fiecare proces trebuie să menţină o coadă de mesaje, *mesQueue*, şi un ceas logic *logicClock*. Pentru a simula execuţia unei operaţii Down sau Up fiecare proces

BUPT

trebuie să transmită broadcast un mesaj la toate nodurile, inclusiv lui însuşi; mesajul conţine identitatea emiţătorului, tipul operaţiei (Down sau Up) şi marca de timp, actualizată conform regulilor prezentate la începutul acestui capitol. Cînd un proces recepţionează un mesaj Down sau Up, îl memorează în coada de mesaje proprie. Aceasta coadă se ordonează crescător, în raport de mărcile de timp conţinute în mesaje iar pentru mărci de timp egale se foloseşte identitatea emiţătorului. Deoarece operaţia de transmisie broadcast nu este atomică, există posibilitatea ca două mesaje broadcast emise de două procese diferite să fie recepţionate în ordine diferită de celelalte procese. Mesajele broadcast emise însă de acelaşi proces vor fi recepţionate de celelalte procese întotdeauna în ordinea crescatoare a mărcilor lor de timp; aceasta oferă posibilitatea luării unor decizii corecte pentru sincronizare. Astfel, presupunînd că procesul conţine în *mesQueue* un mesaj *m* cu marca de timp TS, atunci după ce procesul *i* a recepţionat un mesaj cu o marcă de timp mai mare decit TS, de la toate celelalte noduri, este sigur că nu va mai recepţiona un mesaj cu o marcă de timp mai mică; se spune că mesajul *m* este ***confirmat complet***. În acest caz, orice alt mesaj din coadă, situat înaintea lui *m* va fi, de asemenea confirmat complet deoarece are o marcă de timp mai mică decît *m*. Se formează astfel în coada *mesQueue* un *prefix stabil*: nici un alt mesaj nu va mai fi inserat în această parte a cozii.

Avînd în vedere observaţiile de mai sus, se poate descrie acum algoritmul de implementare a semafoarelor distribuite:

(1) la execuţia unei operaţii Up sau Down, un proces transmite un mesaj broadcast care conţine identitatea sa, tipul operaţiei şi valoarea ceasului logic local.

(2) la recepţionarea unui mesaj Down sau Up, un proces transmite broadcast un mesaj de confirmare (ACK). Confirmările se transmit broadcast pentru a putea fi văzute de toate celelalte noduri. Un mesaj de confirmare conţine marca de timp locală, dar, spre deosebire de mesajele Down şi Up, nu se memorează în *mesQueue*; scopul lor este numai de a determina cînd un mesaj Up sau Down este complet confirmat.

(3) fiecare proces simulează execuţia operaţiilor Down şi Up, determinate de mesajele situate în prefixul stabil al cozii proprii *mesQueue* astfel: fiecare proces menţine doi contori locali *nUp* şi *nDown* care contorizează numărul de mesaje Up, respectiv Down confirmate complet, respectînd invarianţa semaforului. Cînd un mesaj Up devine complet confirmat, se incrementează contorul nUp. Cînd un mesaj Down devine complet confirmat, se incrementează nDown şi, dacă nUp>nDown, se deblochează apelantul. După incrementarea contoarelor, procesul poate şterge mesajul respectiv, Up sau Down, din coada proprie, *mesQueue*.

Nodurile sistemului distribuit se pot afla în faze diferite de prelucrare ale mesajelor Up şi Down, deoarece mesajele pot să fie complet confirmate în ordine diferită, dar fiecare nod va prelucra mesajele complet confirmate în aceeaşi ordine.

Descrierea algoritmului în notatia SR este următoarea:

```
type kind      enum (DOWN, UP, ACK)       # tipuri de mesaje
chan remoteChan[1:n](kind, sender: int, timeStamp: int, sem: int)  #canal broadcast
chan localChan[1:n](kind, sem: int)       # canal pentru transmiterea cererilor
chan answerChan[1:n](semVal: int)         # canal pentru obtinerea valorii contorului

Client [i: 1..n]::
  var sem: int              # pentru a indica numarul semaforului
  var value: int            # pentru a primi valoarea semaforului de la server
        do true →
                ...
                send localChan[i] (UP, sem)          # incrementare semafor
                receive answerChan[i] ( value)       # citire valoare semafor
                send localChan[i] (DOWN, sem)        # sincronizare
                receive answerChan[i] ( value)       # citire valoare semafor

        od

Server [i: 1..n]::
  var localTimeStamp: int = 0          # marca de timp ce va fi asignata urmatoarei cereri locale
  var nUp[1:n]: int = ([n] 0)          # numarul de operatii UP pentru semafoare
  var nDown[1:n]: int = ([n] 0)        # numarul de operatii DOWN pentru semafoare
  var  knd: kind, timeStamp: int       # folosite la citirea mesajelor
  var  sender: int, sem: int           # folosite la citirea mesajelor
  var mesQueue: queue of (sender: int, knd: kind, ts: int, sem: int)

  do true ->                           # bucla infinita a severului
        if not (empty (localChan[i])) →
```


BUPT

```
            receive localChan[i](knd, sem)
            broadcast remoteChan (knd, i, localTimeStamp, sem )
            localTimeStamp := localTimeStamp + 1
    fi
    if not (empty(remoteChan[i])) →
            receive remoteChan[i](knd, sender, timeStamp, sem )
            localTimeStamp := max(localTimeStamp, timeStamp +1)
            localTimeStamp := localTimeStamp + 1
            if knd = UP or knd= DOWN→
                    insereaza in coada mesQueue cererea (sender, kind, timeStamp, sem)
                    broadcast remoteChan[i](ACK, i, timeStamp, evCounter)
                    localTimeStamp := localTimeStamp + 1
            [] knd = ACK
                    inregistreaza confirmarea primita
                    fa cererile UP complet confirmate →
                            inregistreaza identitatea semaforului in sem si a emitatorului in sender
                            sterge cererea din mesQueue
                            nUp[sem] := nUp[sem] +1
                            if sender = i → send answerChan[i](nUp[sem] - nDown[sem])
                    af
                    fa (cererile DOWN complet confirmate) and (nUp[sem] > nDown[sem]) →
                            inregistreaza identitatea semaforului in sem si a emitatorului in sender
                            sterge cererea din mesQueue
                            nDown[sem] := nDown[sem] +1
                            if sender = i → send answerChan[i](nUp[sem] - nDown[sem])
                    af
            fi
    fi
od
```

Există cîte un proces *Client* şi *Server* în fiecare nod. Procesele *Client* iniţiază operaţiile Up şi Down transmiţînd cîte un mesaj serverelor lor locale. Procesele server implementează operaţiile Up şi Down. Ele recepţionează mesajele transmise pe *remoteChan*, gestionează cozile de mesaje, transmit broadcast cererile de operaţii Up sau Down şi confirmările, deblochează procesele *Client* în aşteptare la o operaţie Down şi gestionează ceasul logic local care furnizează mărcile de timp pentru mesaje.

Semafoarele distribuite pot fi utilizate pentru sincronizarea proceselor în programare distribuită într-un mod asemănator cu utilizarea lor în sistemele centralizate. De exemplu, semafoarele distibuite pot fi utilizate pentru realizarea excluderii reciproce în accesul la fişiere, baze de date, pentru a coordona accesul la fişiere replicate. În anexa C se prezinta un exemplu de aplicaţie implementată în reţele Unix şi Microsoft Windows bazată pe semafoarele distribuite.

4.3.4.2. Contori de evenimente distribuiti

O implementare a contorilor de evenimente poate fi obţinută într-un sistem distribuit şi folosind mesaje broadcast. Ca şi implementarea semafoarelor în sisteme distribuite, prezentată la 4.3.4.1., această soluţie de implementare a contorilor de evenimente este complet distribuită. Ideea de bază este dé a transmite mesaje broadcast pentru fiecare operaţie *Advance*; operaţiile *Await* şi *Read* sunt procesate local. Operaţiile *Advance* trebuie confirmate prin mesaje broadcast, de către fiecare nod. Ca şi implementarea semafoarelor distribuite, fiecare proces menţine local o coadă de mesaje incomplet confirmate, *mesQueue*, ordonată crescător în raport de mărcile de timp ale mesajelor şi valoarea contorilor.

Algoritmul de implementare a contorilor de evenimente distribuiţi poate fi descris astfel:

(1) la execuţia unei operaţii *Advance*, un proces transmite un mesaj broadcast care conţine identitatea sa şi valoarea ceasului logic local.

(2) la recepţionarea unui mesaj *Advance*, un proces transmite un mesaj de confirmare (ACK), care conţine valoarea cesului logic local. Scopul mesajelor de confirmare este de a se putea determina cînd un mesaj Advance este complet confirmat.

(3) cînd un mesaj *Advance* (plasat în *mesQueue*) devine complet confirmat, se incrementează contorul şi se elimină mesajul din coadă.

BUPT

(4) Operațiile *Read* și *Await* se procesează local, în nodul respectiv. La execuția operației **Read**(*contor*) se returnează valoarea contorului din momentul respectiv; la execuția operației **Await**(*contor, waitValue*) se deblochează apelantul numai cînd valoarea contorului este mai mare sau egală cu argumentul *waitValue*.

Descrierea algoritmului în notație SR este următoarea:

```
type kind = enum (ADVANCE, READ, AWAIT, ACK)      # tipuri de mesaje
chan remoteChan[1:n](kind, sender: int, timeStamp: int, counter: int) #canal broadcast
chan localChan[1:n](kind, counter: int, case kind of
                                      ADVANCE, READ: ()
                                      AWAIT: waitValue: int
                     )                # canal pentru transmiterea cererilor
chan  answerChan[1:n](counterVal: int)  # canal pentru obtinerea valorii contorului

Client [i: 1..n]::
  var counter: int            # pentru a indica numarul contorului
  var value: int              # pentru a primi valoarea contorului de la server
  var waitValue: int          # pentru a indica valoarea pentru AWAIT
      do true →
          ...
          send localChan[i] (ADVANCE, counter)           # incrementare contor
          send localChan[i] (READ, counter)              # citire valoare contor
          receive answerChan[i] ( value)
          send localChan[i] (AWAIT, counter, waitValue)  # sincronizare

      od

Server [i: 1..n]::
  var localTimeStamp: int = 0          # marca de timp ce va fi asignata urmatoarei cereri locale
  var evCountValues[1:n]: int = ([n] 0)   # valoarea curenta a contorilor
  var evWaitValues[1:n]: int = ([n] -1)   # valoarea curenta a parametrilor transmisi pentru sincronizare
  var  knd: kind, timeStamp: int        # folosite la citirea mesajelor
  var  sender: int, evCounter: int      # folosite la citirea mesajelor
  var mesQueue: queue of (sender: int, ts: int, counter: int) #coada de cereri incomplet confirmate

  do true ->                            # bucla infinita a severului
      if not (empty (localChan[i])) →
            receive localChan[i](knd, evCounter)
                  if knd = ADVANCE →
                        broadcast remoteChan(ADVANCE, i, localTimeStamp, evCounter )
                        localTimeStamp := localTimeStamp + 1;
                  if knd = READ →
                        send answer[i](evCountValues[evCounter])
                  if knd = AWAIT →
                        receive localChan[i](waitValue )
                        evWaitValues[evCounter] := waitValue
                        if  → evCountValues[evCounter] ≥ evWaitValues[evCounter]
                              evWaitValues[evCounter] := -1
                              send  enterChan[i]()
                        fi
                  fi
      fi
      if not (empty(remoteChan[i])) →
            receive remoteChan[i](knd, sender, timeStamp, evCiounter )
            if knd = ADVANCE →
                  insereaza in coada  mesQueue cererea (sender, knd, evCounter)
                  localTimeStamp := max(localTimeStamp, timeStamp +1)
                  localTimeStamp := localTimeStamp + 1
                  broadcast remoteChan[i]( ACK, i, timeStamp, evCounter)
```


BUPT

```
                        localTimeStamp =  localTimeStamp + 1
                [] knd = ACK
                        inregistreaza confirmarea primita
                        fa cererile ADVANCE complet confirmate→
                                inregistreaza identitatea contorului in evCounter
                                sterge cererea din mesQueue
                                evCountValues[evCounter] := evCountValues[evCounter] +1
                                if evWaitValues[evCounter] > 0  and
                                   ( evCountValues[evCounter] ≥ evWaitValues[evCounter])→
                                        evWaitValues[evCounter] := -1
                                        send  enterChan[i]()
                                fi
                        fa
        fi
od
```

Există cîte un proces *Client* şi *Server* în fiecare nod. Procesele *Client* comunică numai cu procesele *Server* locale prin intermediul canalului *localChan*. Procesele *Server* implementează operaţiile *Advance, Read şi Await*. Ele recepţionează mesaje pe canalul *localChan* (de la user-ul asociat) şi pe canalul *remoteChan* (de la celelalte servere) şi asigură transmisia broadcast a mesajelor *Advance*, recepţia mesajelor broadcast (*Advance* şi de confirmare), gestiunea cozilor de mesaje şi deblocarea proceselor *Client* în aşteptare la o operaţie *Await*.

4.3.4.3. Secvenţiatori distribuiţi.

Implementarea complet distribuită a secventiatorilor poate fi facută analog semafoarelor distribuite şi contorilor de eveniment distribuiti, prin utilizarea mesajelor broadcast. În cazul secvenţiatorilor există o singură operaţie (primitivă) care trebuie implementată: primitiva *Ticket()*, care returnează valoarea curentă a secvenţiatorului.

Implementarea presupune şi în acest caz două procese în fiecare nod: procesul *Client* care transmite cererea de execuţie a unei operaţii Ticket la procesul *Server* corespunzător, prin intermediul unui canal local. Operaţia Ticket este transmisă apoi mai departe, la procesele *Server* din toate nodurile, prin transmisia unui mesaj broadcast. Incrementarea valorii secvenţiatorului se face numai pentru operaţiile Ticket complet confirmate.

Descrierea algoritmului în notatia SR este următoarea:

```
type kind  =  enum (TICKET. ACK)                             # tipuri de mesaje
chan  remoteChan[1:n](kind, sender: int, timeStamp: int)     #canal broadcast pentru comunicatie in retea
chan  localChan[1:n](kind)                 # canal pentru transmiterea cererilor de obtinere a unui ticket
chan  answerChan[1:n](seqValue: int)       # canal pentru obtinerea valorii ticketului

Client [i: 1..n]::
var value: int
        do true →
                send localChan[i] (TICKET)          # protocol de obtinere a unui ticket
                receive answerChan[i] (value)

        od

Server [i: 1..n]::
  var  localTimeStamp: int = 0                    # marca de timp ce va fi asignata urmatoarei cereri locale
  var ticketValue: int = 0                        # valoarea curenta a ticketului
  var   knd: kind, timeStamp: int, sender: int, ts: int  # folosite la citirea mesajelor
  var  mesQueue: queue of  (sender: int, ts: int) # coada de mesaje incomplet confirmate

  do true ->                                      # bucla infinita a severului
        if not (empty (localChan[i])) →
                receive localChan[i](knd)
                if knd = TICKET →
```


BUPT

```
                    broadcast remoteChan(TICKET, i, localTimeStamp )
                    localTimeStamp := localTimeStamp + 1;
              fi
        fi
        if not (empty(remoteChan[i])) →
              receive remoteChan[i](knd, sender, timeStamp )
              if knd = TICKET →
                    insereaza in mesQueue cererea
                    localTimeStamp := max(localTimeStamp, timeStamp +1)
                    localTimeStamp := localTimeStamp + 1
                    broadcast remoteChan[i]( ACK, i, timeStamp)
                    localTimeStamp = localTimeStamp + 1
              || knd = ACK
                    inregistreaza confirmarea primita
                    fa cererile TICKET complet confirmate →
                          sterge cererea din coada
                          ticketValue := ticketValue + 1
                          if sender = i → send answer[i](ticketValue) fi
                    fa
              fi
        fi
od
```

4.3.4.3. Implementarea algoritmilor broadcast în rețele Unix și Microsoft Windows (Win32)

Implementarea canalului broadcast (*remoteChan* din descrierea SR a algoritmilor) s-a realizat prin setarea unui soclu UDP ca soclu broadcast, utilizîndu-se *setsockopt()*. Procesele server recepționează mesaje pe două socluri: *localChan*, soclu UDP local, pe care se primesc cereri de la clienții locali și *remoteChan*, soclu UDP pe care se primesc mesajele broadcast trimise de celelalte servere. Toate implementările (pentru semafoare distribuite, contori de evenimente distribuiți și secvențiatori distribuiți) utilizează o listă de mesaje, *mesQueue* în care se memorează cererile clienților incomplet confirmate, ordonată după valoarea mărcilor de timp ale mesajelor; în momentul în care o cerere a unei operații devine complet confirmată (constatare care se face examinînd mesajele de confirmare primite pentru fiecare cerere), cererea respectivă este scoasă din lista.

În cazul algoritmului pentru semafoare distribuite s-a considerat necesar ca, în urma unei operații Up sau Down, serverul să returneze clientului și valoarea asociată semaforului respectiv. Comunicarea între server și client s-a realizat în Windows printr-o zonă de memorie partajată iar în Unix printr-un *pipe*. Sincronizarea dintre client și server s-a realizat în Windows printr-un semafor iar în Unix prin același pipe prin care se realizează și comunicarea valorii semaforului. Analog se realizează comunicarea și sincronizarea server-client și în cazul contorilor de eveniment distribuiți (prin operațiile *Read* și *Await*) și în cazul secvențiatorilor distribuiți (operația *Ticket*).

În cazul algoritmului pentru semafoare distribuite și a contorilor de eveniment distribuiți s-a implementat un tablou de semafoare, respectiv un tablou de contori, astfel că fiecare cerere de operație va avea ca parametru și numărul semaforului, respectiv numărul contorului.

La dispoziția unui program client sunt puse următoarele funcții de bibliotecă:

A. pentru semafoarele distribuite:

1. **int SemaphoreInitialize(void)** pentru crearea și atașarea la zona de memorie partajată, deschiderea semafoarelor, deschiderea soclurilor. Funcția returnează un cod de eroare.
2. **int SemaphoreUp(int sem)** pentru execuția unei operații Up asupra semaforului *sem*. Funcția returnează valoarea semaforului sau -1 dacă *sem* nu este un semafor valid.
3. **int SemaphoreDown(int sem)** pentru execuția unei operații Down asupra semaforului *sem*. Funcția returnează valoarea semaforului din momentul deblocării clientului sau -1 dacă *sem* nu este un semafor valid.
4. **void SemaphoreEnd(void)** pentru deconectarea de la sistem.

B. pentru contorii de eveniment distribuiti:

1. **int EventCountersInitialize(void)** pentru crearea și atașarea la zona de memorie partajată deschiderea semafoarelor, deschiderea soclurilor. Funcția returnează un cod de eroare.
2. **int EventCoutersRead(int counter)** pentru determinarea valorii contorului de eveniment indicat de parametru sau -1 dacă *counter* nu este valid.
3. **int EventCoutersAdvance(int counter)** pentru execuția unei operații Advance(counter).

BUPT

4. **int EventCoutersAwait(int couter, int val)** pentru execuţia unei operaţii Await(couter, val).
5. **void EventCoutersEnd(void)** pentru deconectarea de la sistem.

C. pentru secvantiatorii distribuiţi:

1. **int SequencerInitialize(void)** pentru iniţializari.
2. **void SequencerTicket(void)** pentru obţinerea unui ticket.
3. **void SequencerEnd(void)** pentru deconectarea de la sistem.

Ca exemple, au fost elaborate şase programe client care ilustrează, interogativ, prin intermediul unor interfeţe grafice XWindows, respectiv Windows, modul de lucru cu semafoarele distribuite, contorii de eveniment distribuiţi şi secvenţiatorii. Un alt program client a fost scris pentru a demonstra folosirea atît a contorilor de eveniment cît şi a secvenţiatorilor într-un singur program, care controlează excluderea reciprocă folosind contori şi secvenţiatori (anexa B).

4.4. Algoritmi de alegere

Numeroşi algoritmi pentru sisteme distribuite necesită un mecanism prin care să se desemneze un proces coordonator sau un proces iniţiator care să realizeze funcţii necesare tuturor proceselor din sistem. Aceste funcţii pot fi, de exemplu, un mecanism centralizat de sincronizare, gestiunea unui grafic global pentru detecţia interblocării sau controlul unui dispozitiv de intrare/ieşire. Dacă procesul coordonator dispare datorită unor avarii a nodului sau, sistemul trebuie să-şi continue execuţia prin restartarea unei noi copii a coordonatorului pe un alt procesor disponibil. Algoritmii care determină unde trebuie restartată o noua copie a coordonatorului se numesc *algoritmi de alegere*.

Se consideră, în general, că, pentru simplificare, algoritmii de alegere presupun: (1) o corespondenţă unu la unu între procese şi procesoare; (2) fiecare proces are asociată o prioritate; se presupune că procesorul P_i are prioritatea i; (3) coordonatorul este întotdeauna procesul cu numărul de prioritate cel mai mare; (4) cînd procesul coordonator dispare, algoritmul trebuie să aleagă procesul activ cu cel mai mare număr de prioritate. Acest număr trebuie apoi cunoscut de toate procesele din sistem. În plus, un algoritm de alegere trebuie să furnizeze şi posibilitatea ca un proces care se alatură sistemului să identifice coordonatorul curent.

În acest subcapitol vor fi prezentaţi doi algoritmi de alegere: primul este aplicabil în sisteme în care un proces poate trimite mesaje la toate celelalte procese iar al doilea pentru sisteme în care procesele sunt organizate în inel. Ambii algoritmi necesită n^2 mesaje pentru alegere.

4.4.1. Algoritmul Bully

Presupunem că un proces P_i trimite o cerere la care coordonatorul nu răspunde într-un interval de timp T; în această situaţie, P_i presupune că procesul coordonator a dispărut, şi încearcă să aleagă noul coordonator.

Algoritmul următor, datorat lui Garcia-Molina ([SP88][GMo82]) propune următoarea desfăşurare a procedurii de alegere:

(1) Procesul P_i trimite un mesaj ELECTION la fiecare proces cu un număr de prioritate mai mare ca al său;

(2) P_i aşteaptă apoi răspunsurile de la aceste procese un interval de timp T; dacă nu se primeşte nici un răspuns de la aceste procese se presupune că toate au disparut şi P_i trebuie să devină noul coordonator. Procesul P_i restartează o nouă copie a coordonatorului şi trimite un mesaj INFORM pentru a informa toate procesele active (cu prioritatea mai mică decît a sa) cine este noul coordonator.

(3) Dacă se primeste totuşi un mesaj în intervalul T, P_i aşteaptă încă un interval de timp T' recepţia unui mesaj INFORM (alt proces, mai prioritar s-a ales coordonator). Dacă nu se primeşte mesajul INFORM în intervalul T', P_i presupune că procesul mai prioritar a eşuat în alegere şi restartează algoritmul.

Rezultă că, în orice moment, în timpul execuţiei, P_i poate recepţiona următoarele două mesaje de la un proces P_j: (1) INFORM: P_j este noul coordonator (j>i); P_i memorează această informaţie; (2) ELECTION: P_j a început procedeul de alegere (j<i); P_i trimite un răspuns la P_j şi începe propria sa procedură de alegere. Procesul care termină procedura de alegere va fi cel cu numărul de prioritate cel mai mare şi noul coordonator. La revenirea în sistem a unui proces, se execută din nou acelaşi algoritm. Dacă nu există procese active cu numere de prioritate mai mari, procesul nou intrat forţează toate procesele cu numere mai mici de prioritate să-l accepte ca nou coordonator, chiar dacă există un coordonator activ.

Un exemplu de execuţie al acestui algoritm este prezentat în figura 4.4.1.

BUPT

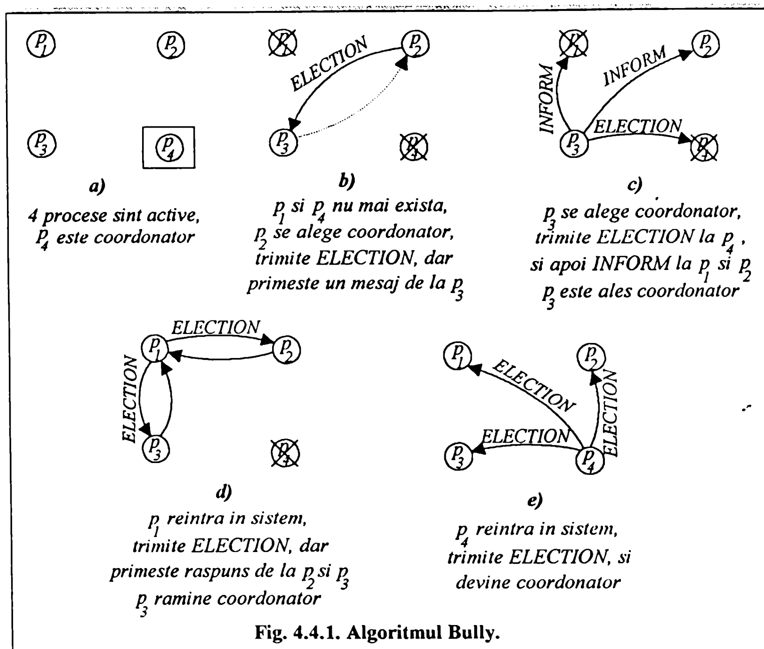


Fig. 4.4.1. Algoritmul Bully.

4.4.2 Algoritm în inel

Tanenbaum descrie în [Tan96][Tan93] următorul algoritm în inel pentru alegerea coordonatorului:
(1) Cînd un proces P_i detectează dispariția coordonatorului, construiește un mesaj ELECTION(i) conținînd numărul procesului pe care îl trimite vecinului din dreapta (succesorul în inel). Dacă succesorul a dispărut, mesajul se trimite la succesorul succesorului s.a.m.d.
(2) La fiecare pas, procesul emițător isi înscrie numărul în mesaj.
(3) Cînd procesul inițiator primește mesajul ELECTION în care și-a înscris numărul, schimbă tipul mesajului în COORDINATOR.
(4) Mesajul COORDINATOR circulă o dată în întreg inelul pentru a informa toate procesele despre identitatea coordonatorului (cel mai mare număr din lista din mesaj) și despre identitatea proceselor din sistem.
Algoritmul se încheie cînd mesajul COORDINATOR a circulat o dată prin tot inelul.

BUPT

Capitolul 5.

PROIECTAREA ŞI IMPLEMENTAREA UNOR MECANISME SUPORT PENTRU DEZVOLTAREA DE APLICAŢII DISTRIBUITE ÎN REŢELE UNIX, MICROSOFT WINDOWS (WIN32) ŞI NOVELL NETWARE

În acest capitol se prezintă proiectarea şi implementarea mai multor variante pentru două toolkit-uri avînd următoarele două scopuri:
(1) c133ontrolul accesului concurent la o bază de date în reţele locale Unix, Windows sau Novell NetWare;
(2) comunicaţii de grup în reţele locale Unix, Windows sau Novell NetWare.

5.1. Proiectarea unui toolkit pentru controlul accesului concurent la o bază de date în reţele locale Unix, Windows (Win32) sau Novell NetWare

Una dintre cele mai importante probleme de sincronizare este coordonarea accesului mai multor utilizatori la un obiect comun, în mod uzual o baza de date, a cărei structură internă nu este specificată, dar care trebuie accesată fie pentru a fi citită, fie pentru a fi modificată [Nut92][Gos91]. Există deci două categorii de utilizatori: utilizatorii care modifică obiectul şi deci, care pe parcursul modificării trebuie să aibă acces exclusiv şi utilizatorii care citesc obiectul şi care, pe parcursul citirii, pot partaja obiectul cu alţi utilizatori în citire.

Există două variante ale problemei: varianta în care utilizatorii care au acces în citire sunt mai prioritari şi varianta în care utilizatorii care au acces exclusiv pentru scriere sunt mai prioritari.

În acest subcapitol vor fi date soluţii pentru ambele variante ale problemei, aceste soluţii se vor implementa în reţele Unix, Windows şi Novell.

Vor fi prezentate două categorii de soluţii: soluţii bazate pe semafoare şi soluţii bazate pe un proces central, coordonator. Semafoarele sunt disponibile în sistemul de operare Novell NetWare iar pentru reţelele Unix şi Windows vor fi utilizate semafoarele distribuite a căror proiectare şi implementare a fost prezentată în subcapitolul 4.3.4.

5.1.1. Soluţii bazate pe semafoare

Soluţiile bazate pe semafoare pleacă de la ideea că, sincronizarea proceselor care cer acces exclusiv, pe de o parte, şi pe de altă parte, sincronizarea lor cu procesele care cer acces partajat se poate face numai cu semafoare.

Starea sistemului poate fi descrisă prin următoarele elemente: (1) doi contori globali unul care contorizează procesele ce sunt în citire la un moment dat (acces partajat) şi denumit *noSharedAccess* şi al doilea contor care contorizează procesele care fie aşteaptă pentru scriere, fie sunt în scriere denumit *noExclAccess*; (2) o variabilă logică, *inExclAccess* care indică dacă o operaţie de scriere este în curs de desfăşurare; (3) două liste: o listă a proceselor aflate în aşteptare pentru citire, *sharedAccessList* şi o listă a proceselor aflate în aşteptare pentru scriere, *exclAccessList*. Pentru lucrul cu aceste liste se presupune că se dispune de două rutine care inserează un proces *p* în listă respectiv extrage un proces *p* din listă *GetList(p,lista)*, *PutList(p,lista)* şi o funcţie care testează dacă o listă este vidă sau nu *Empty(lista)*.

Trebuie interzisă apariţia situaţiilor în care: (1) mai mult decât un proces este în acces exclusiv în scriere; (2) un proces este în scriere, în timp ce altul este în citire. Prima condiţie va fi îndeplinită implicit datorită modului în care au fost alese variabilele de stare; considerînd cazul în care procesele care cer acces exclusiv sunt mai prioritare, a doua condiţie poate fi exprimată prin (not *inExclAccess* or *NoSharedAccess* 0). La sfîrşitul unei operaţii de scriere sau al unui ansamblu de operaţii de citire dacă ambele liste de procese nu sunt vide, se va da prioritate proceselor în aşteptare la scriere.

În general, procesele vor avea structura următoare:

BUPT

```
ReadProcess[i:1..n]::
        do true →
                OpenSharedAccess(i)
                #citirea bazei de date
                CloseSharedAccess(i)
        od
WriteProcess[i:1..n]::
        do true →
                OpenExclAccess(i)
                #scriere in baza de date
                CloseExclAccess(i)
        od
```

Se vor prezenta, în continuare, trei soluţii de codificare a rutinelor *OpenSharedAccess, CloseSharedAccess, OpenExclAccess şi CloseExcAccess* folosind semafoare.

A. Prima soluţie bazată pe semafoare

Pentru a controla accesul la variabilele de stare va fi introdus un semafor care asigură excluderea reciprocă a celor *n* procese în accesul la aceste variabile, *SharedDataProtect*. Fiecrui proces *k* i se asociaza un semafor *WaitingAcces[k]* pentru a controla accesul la baza de date. Ţinînd cont şi de semnificaţiile variabilelor de stare, descrierea rutinelor *OpenSharedAccess, CloseSharedAccess, OpenExclAccess şi CloseExcAccess* este următoarea:

```
var NoSharedAccess: int :=0;      # contorizeaza numarul de procese in acces partajat
var NoExclAccess: int :=0;        # contorizeaza numarul de procese in asteptare pentru acces exclusiv si in
                                  #acces exclusiv
var InExclAccess: bool :=false    #indica daca un proces este in cursul unui acces exclusiv
var ExclAccessList, SharedAccessList: queue of processes :=∅, ∅   # lista de procese
var SharedDataProtect: semaphore := 1          #controleaza accesul la variabilele de stare
var WaitingAccess[1:n]: semaphore :=([n] 0)    #coordoneaza blocarea si deblocarea proceselor

OpenSharedAccess[i:1..n]::
        Down(SharedDataProtect) # se protejeaza actualizarea variabilelor de stare
        if NoExclAccess = 0 →
                NoSharedAccess:= NoSharedAccess + 1     # inca un proces in acces partajat
                Up(WaitingAccess[i])                    # se pregateste validarea accesului
        [] NoExclAccess ≠ 0 → PutList(SharedAccessList,i) # se intirzie citirea din baza de date
        fi
        Up(SharedDataProtect)    # se deblocheaza variabilele de stare
        Down(WaitingAccess[i])   # se incearca intrarea in acces partajat
        return

CloseSharedAccess[i:1..n]
        Down(SharedDataProtect)                   # se protejeaza actualizarea variabilelor de stare
        NoSharedAccess:= NoSharedAccess - 1       # s-a terminat un proces care a citit baza de date
        if NoSharedAccess=0 and not empty(ExclAccessList) →     #nici un proces nu citeste baza de date
                GetList(ExclAccessList,k) # se extrage din lista un proces care asteapta pentru scriere
                InExclAccess:= true       # se marcheaza intrarea in acces exclusiv
                Up(WaitingAccess[k])      # se deblocheaza procesul k pentru a intra in acces exclusiv
        fi
        Up(SharedDataProtect)             # se deblocheaza variabilele de stare
        return

OpenExclAccess[i:1..n]
        Down(SharedDataProtect)          # se protejeaza actualizarea variabilelor de stare
        NoExclAccess:= NoExclAccess + 1 # inca un proces care asteapta pentru scriere
        if NoSharedAccess=0 and not InExclAccess → # nu exista nici un proces in citire sau scriere
                InExclAccess:= true        # se marcheaza accesul exclusiv
```

```
            Up(WaitingAcces[i])          # se pregateste validarea accesului
      [] NoSharedAccess ≠ 0 →
            PutList(ExclAccessList, i) # se intirzie scrierea
      fi
      Up(SharedDataProtect)            # se deblocheaza variabilele de stare
      Down(WaitingAccess[i])           # se incearca intrarea in acces partajat
      return

CloseExclAccess[i:1..n]::
      Down(SharedDataProtect)          # se protejeaza actualizarea variabilelor de stare
      NoExclAccess := NoExclAccess - 1 # s-a terminat un proces care a scris in baza de date
      InExclAccess:=false              # se marcheaza iesirea din acces exclusiv
      if not empty(ExclAccessList) →   # daca exista procese care asteapta pentru scriere
            GetList(ExclAccessList,k)  # se extrage din lista un proces care asteapta pentru scriere
            InExclAccess:= true        # se marcheaza intrarea in acces exclusiv
            Up(WaitingAccess[k])       # se deblocheaza procesul k pentru a intra in acces exclusiv
      [] empty(ExclAccessList) →
            do not empty(SharedAccessList) →       #cit timp exista procese in asteptare pentru citire
                  GetList(SharedAccessList,k)      # se extrage un proces din lista
                  NoSharedAccess:= NoSharedAccess + 1   # se contorizeaza
                  Up(WaitingAccess[k])             # si se deblocheaza pentru a intra in citire
            od
      fi
      Up(SharedDataProtect)            # se deblocheaza variabilele de stare
      return
```

Se observă că un proces poate trece de **Down(** *WaitingAccess*[*k*]) dacă fie baza de date a fost liberă, fie a fost activat de un alt proces in *CloseExclAccess()* sau *CloseSharedAccess()*. Toate procesele în aşteptare pentru scriere vor fi activate, pe rind, la sfîrşitul fazelor de scriere, ele avînd prioritate faţă de procesele în aşteptare pentru citire. Procesele în aşteptare pentru citire vor fi activate la sfîrşitul unei faze de scriere numai dacă lista *ExclAccessList* este vidă. Listele conţin identificatorii proceselor în aşteptare; rutinele GetList() şi PutList() pot fi realizate pentru a implementa o anumită strategie de planificare a proceselor.

B. A doua soluţie cu semafoare

Soluţia prezentată mai sus poate fi uşor adaptată pentru diverse strategii de planificare a proceselor. În schimbul obţinerii acestui avantaj, trebuie manţinute, însă, explicit, liste de procese, *ExclAccessList* şi *SharedAccessList*. Cu costul micşorării flexibilităţii se poate renunţa la gestionarea explicită a listelor de procese dacă se fac urmatoarele observaţii: (1) fiecărui set de secţiuni critice îi corespunde implicit o listă de procese în aşteptare (2) un set de secţiuni critice pot fi blocate, în mod selectiv, astfel incît mai mult decît un proces dintr-o anumită clasă poate avea acces simultan la baza de date sau un singur proces din altă clasă să aibă acces exclusiv. Mecanismul de blocare selectivă utilizează următoarele două operaţii:

```
DownWithCondition(exclusive, waiting: semaphore; var counter: int)::
      Down(exclusive)
      if counter = 0 → Down(waiting) fi
      counter:= counter + 1
      Up(exclusive)
      return
UpWithCondition(exclusive, waiting:semaphore; var counter: int)::
      Down(exclusive)
      counter:= counter + 1
      if counter = 0 → Up(waiting) fi
      Up(exclusive)
      return
```

Operaţia *DownWithCondition*(*exclusive, waiting, counter*) va bloca o operaţie Down(*waiting*); totuşi apelul următor *DownWithCondition*() cu aceeaşi parametri nu va produce blocare. Operaţia *UpWithCondition*(*exclusive, waiting, counter*) este operaţia corespondentă de deblocare.

Utilizînd aceste operaţii, codificarea rutinelor de intrare şi ieşire din secţiunile critice este:

BUPT

```
var NoSharedAccess: int :=0            # contorizeaza numarul de procese in acces partajat
var NoExclAccess: int :=0              # contorizeaza numarul de procese in asteptare pentru acces
                                       # exclusiv si in acces exclusiv
var SharedDataProtect: semaphore :=1   # folosit pentru protectia datelor in operatiile
                                       # DownWithCondition() si UpWithCondition()
var Exclusive: semaphore :=1 # folosit pentru accesul exclusiv la datele partajate de procesele care cer citire
var ExclDataProtect: semaphore :=1     # folosit pentru protectia datelor in operatiile
                                       # DownWithCondition si UpWithCondition
var StopExclAccess: semaphore :=1      # folosit pentru blocarea proceselor care cer scriere
var StopSharedAccess: semaphore :=1    # folosit pentru blocarea proceselor care cer citire

OpenSharedAccess[i:1..n]::
        Down(Exclusive)
        Down(StopSharedAccess)
        DownWithCondition(SharedDataProtect, StopExclAccess, NoSharedAccess)
        Up(StopSharedAccess)
        Up(Exclusive)
        return

CloseSharedAccess[i:1..n]
        UpWithCondition(SharedDataProtect, StopExclAccess, NoSharedAccess)
        return

OpenExclAccess[i:1..n]
        DownWithCondition(ExclDataProtect, StopSharedAccess, NoExclAccess)
        Down(StopExclAccess)
        return

CloseExclAccess[i:1..n]::
        Up(StopExclAccess)
        UpWithCondition(ExclDataProtect, StopSharedAccess, NoExclAccess)
        return
```

Operaţia *DownWithCondition*() din *OpenSharedAccess*() va genera blocarea unui proces care execută o operaţie Down(*StopExclAccess*) în *OpenExclAccess*() dar nu şi alte procese care execută *DownWithCondition()* în *OpenSharedAccess*(); ca atare, un proces care are acces în citire la baza de date va bloca alte procese care cer scriere, dar nu şi alte procese care cer citire. Operaţia Down(*StopExclAccess*) va bloca alte procese care cer citire

C. A treia soluţie cu semafoare

Se prezintă acum o variantă în care procesele care cer acces partajat sunt mai prioritare. Singurul caz în care un proces care cere acces partajat trebuie să aştepte este acela în care baza de date este ocupată de un proces în scriere. Un proces care cere scriere nu poate conţine accesul exclusiv decît atunci cînd nici un proces care cere citire nu este în execuţie sau aşteptare. Codificarea algoritmului este următoarea:

```
var NoSharedAccess : int := 0          # contorizeaza numarul de procese in acces partajat
var Exclusive : semaphore :=1          # folosit pentru protectia variabilelor comune
var PrioritySharedAccess : semaphore :=1  # folosit pentru a obtine prioritatea proceselor care cer citire
var WaitingExclAccess : semaphore :=1  # folosit pentru a bloca procesele care cer citire

OpenSharedAccess [i:1..n]:
        Down(Exclusive)                            # se protejeaza actualizarea variabilelor de stare
        NoSharedAccess := NoSharedAccess +1        # inca un proces care citeste din baza de date
        if NoSharedAccess = 1 → Down(PrioritySharedAccess) fi   # se blocheaza accesul proceselor
                                                   # care cere scriere
        Up(Exclusive)                              # se deblocheaza variabilele de stare
        return

CloseSharedAccess [i:1..n]::
```


BUPT

```
        Down(Exclusive)
        NoSharedAccess := NoSharedAccess -1       # s-a terminat un proces care e citit din baza de date
        if NoSharedAccess = 0 → Up(PrioritySharedAccess) fi      # se deblocheaza accesul exclusiv
        Up(Exclusive)                  # se deblocheaza variabilele de stare
        return

    OpenExclAccess [i:1..n]::
        Down(WaitingExclAccess)        # se blocheaza alte procese care cer scriere
        Down(PrioritySharedAccess)     # se blocheaza alte procese care cer citire
        return

    CloseExclAccess [i:1..n]::
        Up(PrioritySharedAccess)       # se deblocheaza intii procesele care asteapta pentru citire
        Up(WaitingExclAccess)          # se deblocheaza apoi procesele care asteapta pentru scriere
        return
```

Se observă că: (1) semaforul ***Exclusive*** se foloseşte pentru a proteja incrementarea şi testarea variabilei *NoSharedAccess* (2) un proces care citeşte baza de date blochează alte procese care cer scriere prin Down (*PrioritySharedAccess*) dar nu şi alte procese care cer citire (3) un proces care a obţinut accesul exclusiv blochează alte procese care cer acces pentru scriere la **Down** (*WaitingExclAccess*) şi alte procese care cer acces pentru citire la Down (*PrioritySharedAccess*) (4) Prioritatea proceselor care cer citire se obţine datorită observaţiilor 2 şi 3.

5.1.2. Soluţie bazată pe un proces coordonator

Toate soluţiile de sincronizare prezentate anterior au în comun faptul ca operaţiile pentru obţinerea accesului în sectiunile critice sunt realizate prin apelul unor rutine executate în cadrul proceselor apelate; astfel, dacă un proces este întrerupt în timp ce încearcă să obţina accesul într-o secţiune critică, baza de date devine blocată pe un interval nedefinit. Acest dezavantaj poate fi înlăturat dacă se separă toate operaţiile pentru obţinerea accesului şi se înglobează într-un proces separat faţă de apelant, asociat bazei de date şi denumit manager. O operaţie pentru obţinerea accesului se apelează printr-un mesaj transmis procesului manager, care returnează un mesaj de răspuns atunci cînd apelantul poate intra în secţiune critică.

Codificarea procesului manager poate fi realizat prin analogie cu prima soluţie cu semafoare. În plus faţă de variabilele de stare, se mai menţine o variabila *CliNo* care conţine identificatorul clientului apelant şi o variabila *Request* care identifică tipul operaţiei cerute. În aceste condiţii, procesul manager poate fi codificat astfel:

```
type Kind = enum (r,R,w,W)                   # tipuri de operatii
chan managerChan(kind, cliNo: int)           # canal pentru transmiterea mnesajelor la manager
chan clientChan[1:n] ( )                     # canale ale clientilor pentru primirea raspunsurilor de la manager

Manager::
    var NoSharedAccess: int: = 0             # contorizeaza numarul de procese in acces partajat
    var NoExclAccess:  int :=0               # contorizeaza numarul de procese in asteptare pentru
                                             # acces exclusiv si in acces exclusiv
    var InExclAccess : bool: = false         #indica daca un proces este in cursul unui acces exclusiv
    var ExclAccessList, SharedAccessList: queue of processes : = ∅, ∅  # liste de procese
    var CliNo: int                           # folosit ca identificator de proces
    var Request: kind                        # folosit pentru a identifica tipul operatiei cerute

 OpenSharedAccess()
        if NoExclAccess = 0 →                #indica daca un proces este in cursul unui acces exclusiv
                NoSharedAccess : = NoSharedAccess + 1
                send clientChan[CliNo]() # deblocare apelant
        [] NoExclAccess ≠ 0 → PutList(SharedAccessList, CliNo)
        fi
        return

 CloseSharedAccess()
```


BUPT

```
        NoSharedAccess := NoSharedAccess -1        # s-a terminat un proces care a citit din baza de date
        send ClientChan[CliNo]()                   # se deblocheaza clientul apelant
        if NoSharedAccess = 0  and  not empty (ExclAccessList) →
                GetList(ExclAccessList, CliNo)
                InExclAccess := true
                send ClientChan[CliNo]()           # se deblocheaza un client care a cerut scriere
        fi
        return
OpenExclAccess()
        NoExclAccess := NoExclAccess +1            # un alt proces in asteptare pentru scriere
        if NoSharedAccess = 0 and not InExclAccess →
                InExclAccess := true
                send ClientChan[CliNo]()                      # se deblocheaza clientul
        [] not ( NoSharedAccess = 0 and not InExclAccess) →   # se amina accesul exclusiv
                PutList(ExclAccessList,CliNo)
        fi
        return
CloseExclAccess()
        NoExclAccess : = NoExclAccess -1           # s-a terminat un proces care a scris in baza de date
        InExclAccess := false
        send Client Chan[CliNo]()                  # se deblocheaza clientul apelant
        if not empty (ExclAccessList)  →
                GetList (ExclAccessList,CliNo)
                InExclAccess := true
                send ClientChan [CliNo]()          # se deblocheaza un client care a cerut scriere
        [] empty (ExclAccessList)  →
                do not empty (SharedAccessList)  →
                        GetList (SharedAccessList,CliNo)
                        NoSharedAccess := NoSharedAccess +1
                        send ClientChan[CliNo]()
                od
        fi
        return

Manager()
        do true  →
                receive managerChan(Request,CliNo)        # se primeste un mesaj care contine o cerere de
                                                          # executie a unei operatii
                case Request of
                        r: OpenSharedAccess()
                        R : CloseSharedAccess()
                        w: OpenExclAccess()
                        W: CloseExclAccess()
                esac
        od
```

Procesele client pot fi codificate astfel:

```
Client [i:1..n]::
        do true  →
                send managerChan (r, i)            # protocol pentru intrare in sectiunea critica pentru
                receive clientChan[i]()            # citire. la fel pentru scriere
                # citire din baza de date          # citire
                send managerChan(R, i)             # protocol pentru iesirea din sectiunea critica
                receive client Chan[i]()
        od
```


BUPT

5.1.3. Implementări în reţele Unix, Microsoft Windows (Win32) şi Novell Netware

Soluţiile bazate pe semafoare şi prezentate la 5.1.1 au fost implementate în reţele Novell Netware, avînd în vedere că acest sistem de operare furnizează mecanismul semafoarelor şi în reţele Unix şi Microsoft Windows, avînd în vedere ca în acest sisteme de operare a fost implementat mecanismul semafoarelor distribuite (vezi 4.3.4.1 şi 4.3.4.4.)

5.1.3.1.Implementarea soluţiilor bazate pe semafoare în reţele Novell Netware

Implementarea soluţiilor bazate pe semafoare in reţele Novell Netware este prezentată în anexa C. Ceea ce este caracteristic tuturor acestor trei implementări este faptul că variabilele comune tuturor proceselor (variabilele de stare) au fost implementate utilizind semafoare. Prin urmare, pentru incrementarea contoarelor *NoSharedAccess, NoExclAccess* vor fi executate funcţiile Netware *SignalSemaphore()*, pentru incrementare, respectiv *WaitOnSemaphore()* pentru decrementare, iar pentru examinarea valorilor lor, funcţia *ExamineSemaphore()*. Implementarea soluţiei a doua şi a treia cu semafoare este mai directă, deoarece nu necesită liste de procese. Implementarea primei soluţii necesită realizarea a două liste de procese; s-a optat ca un proces, asociat unui singur nod, să fie identificat prin numărul de conexiune asociat nodului respectiv. Deoarece şi listele de procese sunt variabile comune, este necesar ca ele să fie realizate utilizînd tot semafoare; astfel, pentru implementarea listelor se foloseşte un tablou de semafoare, a cărui dimensiune maximă este egală cu cel mai mare număr de conexiune posibil. Faptul că un proces este în aşteptare pentru scriere va fi semnalat prin două operaţii *SignalSemaphore()* asupra semaforului din tabloul de semafoare de pe poziţia corespunzătoare numărului de conexiune asociat procesului; analog, printr-o singură operaţie *SignalSemaphore()* va fi semnalat un proces care aşteaptă la citire.

Toate trei soluţiile oferă unui program utilizator următoarele funcţii de bibliotecă:

(1) int *ExclOrSharedAccessInitialize*() apelată pentru deschiderea semafoarelor şi determinarea numărului de conexiune propriu. Funcţia returnează un cod de eroare sau 0 pentru execuţie fără erori.
(2) int *OpenSharedAccess*(void) apelată pentru obţinerea accesului in mod partajat.
(3) int *OpenExclAccess*(void) apelată pentru obţinerea accesului in mod exclusiv
(4) int *CloseSharedAccess*(void) apelată pentru terminarea accesului in mod partajat.
(5) int *CloseExclAccess*(void) apelată pentru terminarea accesului in mod exclusiv.
(6) int *ExclOrSharedAccessEnd*(void) apelată pentru deconectarea de la sistem.

Toate funcţiile returnează un cod de eroare sau zero pentru execuţie cu succes.

5.1.3.2 Implementarea soluţiilor bazate pe semafoare în reţele Unix şi MSWindows (Win32)

Implementarea in reţele Unix şi Windows a soluţiei a treia bazată pe semafoare, în care procesele care cer acces partajat sunt mai prioritare, este prezentată in anexa C.

Se foloseste serverul pentru semafoare distribuite a cărui proiectare şi implementare a fost prezentată la 4.3.4.1. şi 4.3.4.4.

Pe baza funcţiilor ***Semaphore*_XX()** puse la dispoziţia programului client au fost concepute, conform algoritmului, funcţiile:

(1) void *OpenSharedAccess*(void) apelată pentru obţinerea accesului in mod partajat.
(2) void *OpenExclAccess*(void) apelată pentru obţinerea accesului in mod exclusiv.
(3) void *CloseSharedAccess*(void) apelată pentru terminarea accesului in mod partajat.
(4) void *CloseExclAccess*(void) apelată pentru terminarea accesului in mod exclusiv.

Esenţial pentru implementarea variabilelor comune prin semafoare distribuite este faptul ca funcţiile ***Semaphore_Up()*** şi ***Semaphore_Down()*** returnează valoarea semaforului din momentul inceperii execuţiei funcţiei.

Pentru a folosi funcţiile de mai sus, un program client (cu sau fără interfaţă grafică) trebuie linkeditat cu aceste funcţii şi, de asemenea trebuie executat in background serverul pentru semafoare distribuite. Semafoarele care simulează variabilele comune sunt iniţializate la inceputul programului prin apelul explicit al funcţiilor Semaphore_Down şi Semaphore_Up.

În varianta prezentată în anexă se lucrează cu un singur client intr-un nod, ataşat unui server pentru semafoare distribuite.

BUPT

5.1.3.3. Implementarea soluţiei bazată pe un proces coordonator în reţele Novell Netware

Protocolul folosit pentru comunicarea clienţi-proces coordonator (server) este protocolul IPX. Identificarea atît a clienţilor cît şi a procesului coordonator se face pe baza numerelor de conexiuni; în orice mesaj emis catre server, clienţii înglobează şi numărul propriu de conexiune, pentru a putea fi identificaţi pentru o deblocare ulterioară.

Serverul construieşte o coadă de cereri *queue[MAX_NO_PACKETS]*; sosirea unui mesaj de la un client declanşează execuţia unei rutine declarate ca EventServiceRoutine, care depune în coada de cereri mesajul sosit. Se ocoleşte astfel situaţia ca, datorită unei supraîncărcari a serverului, acesta să piardă mesaje. Listele de procese sunt implementate ca tablouri de conexiuni ale proceselor client. Funcţia *GetPriProcess*() extrage un proces client din listă conform priorităţii proceselor client, alese egale cu conexiunile lor.

Procesele client li se pun la dispoziţie aceleaşi funcţii ca şi la 5.1.3.1:

(1) int *ExclOrSharedAccessInitialize*() apelată pentru deschiderea soclurilor, conectarea rutinei ESR, determinarea adresei Ethernet a serverului cunoscîndu-se user-ul la care acesta a fost lansat, determinarea conexiunii proprii.

(2) int *OpenSharedAccess*(void) pentru obţinerea accesului în mod partajat, prin schimb de mesaje cu serverul.

(3) int *OpenExclAccess*(void) apelată pentru obţinerea accesului în mod exclusiv.

(4) int *CloseSharedAccess*(void) apelată pentru terminarea accesului în mod partajat.

(5) int *CloseExclAccess*(void) apelată pentru terminarea accesului în mod exclusiv.

(6) int *ExclOrSharedAccessEnd*(void) apelată pentru deconectarea de la sistem.

Pentru demonstrarea tuturor soluţiilor implementate în reţele Novell Netware a fost scris un singur program client care se linkeditează cu setul de funcţii pus la dispoziţie de soluţia aleasă pentru verificare.

5.1.4. Compararea soluţiilor propuse şi concluzii

În urma analizei celor patru soluţii propuse şi a implementării lor în reţele locale se pot enunţa următoarele concluzii:

(1) Toate soluţiile prezentate necesită implementarea unor variabile de stare; acestea pot fi implementate fie separat, în cadrul unui proces coordonator, fie pot fi simulate ca "variabile comune" prin asimilarea lor cu valorile unor semafoare distribuite.

(2) Prin faptul că menţin o listă de procese, prima soluţie cu semafoare şi soluţia procesului central pot planifica procesele pentru execuţie după diverşi algoritmi.

(3) Pe de altă parte, a doua şi a treia soluţie cu semafoare, negestionînd liste, sunt mai rapide.

(4) Implementarea soluţiei bazată pe un proces coordonator se face avînd în vedere raportul dintre timpul necesar tratării unei cereri şi rata de sosire a cererilor, în conformitate cu cele arătate la 3.1.2., pentru a elimina posibilitatea apariţiei situaţiei de "sufocare" a serverului.

(5) Toate soluţiile prezintă posibilitatea apariţiei amînării continue a unui proces din cele două categorii de procese (în funcţie de cui se acordă prioritate: proceselor care cer acces exclusiv sau celor care cer acces partajat). Amînarea la infinit a proceselor poate fi înlăturată dacă, la terminarea unui proces care cere acces exclusiv (cazul primelor două soluţii) se au în vedere ambele liste de procese în aşteptare.

(6) Deoarece mesajele care se transmit în sistem, sunt de lungime foarte mică, avînd semnificaţie numai pentru sincronizare, s-a optat ca mesajele să fie codificate ASCII astfel încît comunicarea este valabilă in reţele eterogene (fără a se pune problema de "big" sau "little "endian") . Singura cerinţă este cea a sistemului de operare (Unix sau Windows).

(7) Avînd în vedere că sistemul Unix este multiuser, o cerinţă în plus ar fi eliminarea restricţiei unui singur client pentru un singur server pe un nod. Restricţia provine de fapt de la implementarea semafoarelor distribuite. Ea poate fi eliminată uşor dacă se foloseşte pentru fiecare client, pentru sincronizare, un canal *pipe* rezervat special, pentru Unix; aceste canale pot fi alocate static, la iniţializarea serverului, dar limitînd astfel numărul de clienţi de pe un nod, sau dinamic, dar în acest caz însă mai trebuie introdusă şi o cerere specială, a fiecărui client, de "conectare" la sistem.

(8) Identificarea clienţilor s-a făcut pe baza numerelor de conexiune în Novell şi pe baza unor identificatori transmişi prin linia de comandă în Unix şi Windows. Dacă sistemul se extinde pentru a permite mai multe procese client într-un nod se poate folosi aceeasi metodă, pentru a identifica canalele *pipe* asociate clienţilor pe baza acestori identificatori sau se pot folosi identificatorii asociaţi proceselor de către sistemul de operare ("pid"-urile Unix ale proceselor).

(9) Toate soluţiile prezentate se bazează pe respectarea protocolului de obţinere a accesului de către procesele client.

BUPT

5.2. Proiectarea şi implementarea unui toolkit pentru comunicaţii de grup

Acest capitol înglobează aspectele teoretice prezentate în capitolele 2,3 şi 4 în cadrul proiectării unui toolkit pentru comunicaţii de grup în reţele Unix şi Windows, denumit TGC.

5.2.1. Cerinţe de proiectare

Înainte de a prezenta arhitectura sistemului TGC se enumeră cateva cerinţe de proiectare impuse:
(1) Sistemul va folosi modelul grupurilor *deschise*; aceasta înseamnă că, pentru a putea transmite mesaje la toate procesele componente ale unui grup, un proces conectat la TCG nu trebuie să fie membru al acelui grup.
(2) Un proces trebuie să poată aparţine simultan mai multor grupuri.
(3) Fiecare grup multicast poate avea asociate mai multe pseudonime (*alias*-uri).
(4) Sistemul trebuie să se execute în mod *utilizator;*
(5) Protocolul de comunicaţie trebuie să fie independent de maşina hardware;
(6) Sistemul trebuie să genereze un trafic de mesaje minim.
(7) Sistemul trebuie să furnizeze o comunicaţie sigură.

TGC trebuie sa furnizeze primitive relative la:
(1) crearea, alăturarea la şi părăsirea unui grup;
(2) atribuirea de pseudonume grupurilor, dezafectarea unor pseudonume, căutarea unui grup;
(3) transmisia şi recepţionarea de mesaje la/de la procese aparţinînd unui grup multicast;
(4) conectarea şi deconectarea de la sistem.

Înainte de apelul oricărei operaţii, orice client trebuie să se conecteze la sistem; conectarea la sistem înseamnă obţinerea unui identificator, care este apoi livrat în corpul fiecărei cereri, pentru identificarea clienţilor.

Se prezintă în continuare o descriere în detaliu a primitivelor furnizate de TGC:

(1) `int CreateGroup(idGroup *pgroup, char *name)`

Scop:
Asigură crearea unui nou grup multicast, al cărui nume se indică în apel; procesul apelant devine automat un membru al grupului. Identificatorul noului grup se stabileşte de către un server central şi este unic; el este returnat apelantului.

Parametri:
`name` -reprezintă numele stabilit pentru noul grup.
`pgroup` - pointer la identificatorul noului grup creat

Valoare returnată:
Funcţia returnează valoarea *SUCCESS* în cazul în care crearea a reuşit, sau un cod de eroare in caz contrar. Codurile de eroare care pot fi returnate sunt:

[*EBADF*]	Format eronat al parametrului de intrare (`nume` indică un şir nul)
[*EGROUP_ALREADY_CREATED*]	Grupul există deja
[*ETOO_MANY_GROUPS*]	Spaţiu insuficient pentru înregistrarea unui nou grup
[*ETOO_MANY_HOSTS_IN_GROUP*]	Grupul este deschis pe maximum de noduri posibil
[*ECLIENT_NOT_CONNECTED*]	Procesul nu s-a conectat la sistem

(2) `int JoinGroup(idGroup group)`

Scop:
Permite unui proces ataşarea la un grup multicast, `group`. Orice proces trebuie să execute această primitivă pentru a putea transmite sau recepţiona mesaje în contul acestui acestui grup.

Parametri:
`group` - identificatorul grupului la care se doreşte ataşarea

Valoare returnată:
Funcţia returnează valoarea *SUCCESS* în cazul în care alăturarea a reuşit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi returnate sunt:

[*ETOO_MANY_CONC_GROUPS*]	Procesul este ataşat la numarul maxim de grupuri
[*EFULL_GROUP*]	Spaţiu insuficient pentru înregistrarea unui nou proces în acest grup
[*ECLIENT_ALREADY_CONNECTED*]	Procesul există deja în grup
[*EWRONG_DEST_GROUP*]	Grupul la care procesul doreşte să se ataşeze nu este creat

BUPT

[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(3) `int LeaveGroup(idGroup group)`

Scop:

Permite părăsirea unui grup grup de către un proces. Primitiva se execută de catre un client în momentul în care nu mai doreşte apartenenţa la un grup. După părăsirea unui grup, procesul respectiv nu mai poate primi mesaje adresate acelui grup, chiar dacă ele sunt memorate deja în buffer-ul de mesaje al acelui proces.

Parametri:

`group` - identificatorul grupului la care procesul aparţine

Valoare returnată:

Funcţia returnează valoarea *SUCCESS* în cazul în care operaţia a reuşit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obţinute sunt:

[*WRONG_DEST_GROUP*] Grupul nu există
[*ECLIENT_NOT_IN_DEST_GROUP*] Procesul nu aparţine grupului
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(4) `int CreateAliasGroup(idGroup group, char *alias)`

Scop:

Permite crearea unui psudonim pentru grupul `grup`; nu sunt permise aceleaşi pseudonume pentru două grupuri. Procesul apelant nu trebuie sa fie membru al grupului.

Parametri:

`group` - identificatorul grupului pentru care se doreşte crearea pseudonumelui
`alias` - pseudonumele

Valoare returnată:

Funcţia returnează valoarea *SUCCESS* în cazul în care operaţia a reuşit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obţinut sunt:

[*EBADF*] Format eronat al parametrului de intrare (`alias` indică un şir nul)
[*WRONG_DEST_GROUP*]Grupul nu există
[*ETOO_MANY_ALIASES*]Spaţiu insuficient pentru înregistrarea unui nou pseudonume
[*WRONG_ALIAS*] Există un grup care are acest pseudonume
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(5) `int DestroyAliasGroup(idGroup group, char *alias)`

Scop:

Permite distrugerea unui pseudonume pentru grupul `grup`. Procesul apelant nu trebuie să fie membru al grupului.

Parametri:

`group` - identificatorul grupului pentru care se doreşte distrugerea pseudonumelui
`alias` - pseudonumele

Valoare returnată:

Funcţia returnează valoarea *SUCCESS* în cazul în care operaţia a reuşit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obţinute sunt:

[*EBADF*] Format eronat al parametrului de intrare (`alias` indică un şir nul)
[*WRONG_DEST_GROUP*] Grupul nu există
[*WRONG_ALIAS*] Pseudonumele nu există pentru acest grup
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(6) `int GetIdGroup(idGroup *pgroup, char* alias)`

Scop:

Permite căutarea identificatorului unui grup cînd se cunoaşte un pseudonume.

Parametri:

`alias` -un pseudonume al grupului pentru care se doreşte obţinerea identificatorului
`pgroup` - la adresa `pgroup` se returnează, în caz de succes, identificatorul grupului.

Valoare returnată:

Funcţia returnează valoarea *SUCCESS* în cazul în care operaţia a reuşit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obţinute sunt:

[*EBADF*] Format eronat al parametrului de intrare (`alias` indică un şir nul)
[*EWRONG_ALIAS*] Pseudonumele nu există

BUPT

[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(7) **int SendToGroup(idGrup** *dest_gr_id*,**idGrup** *source_gr_id*, **char** **mes*, **int** *counter*, **pf** *filter*)

Scop:
Permite transmiterea unui mesaj tuturor proceselor dintr-un grup multicast indicat în parametrul `dest_group`. Deoarece procesul care transmite mesajul poate fi conectat simultan la mai multe grupuri, el trebuie să indice sistemului, în parametrul `source_gr_id`, grupul sursă în contul căruia se transmite mesajul. Mesajul constă într-un șir de octeți de lungime `contor`. În mesajul transmis de sistemul TCG se transmite și numele utilizatorului, indicat la conectarea la sistem prin funcția `InitTCG()`.

Parametri:
`dest_gr_id` - identificatorul grupului la care se dorește transmisia mesajului
`source_gr_id` - identificatorul grupului la care aparține procesul care transmite mesajul, grup în contul căruia se transmite mesajul
`counter` - dimensiunea mesajului
`mes` –. mesajul de transmis
`pfilter` – pointer la o funcție utilizator apelată de sistemul TGC înainte de emisia fiecărui mesaj, pentru codificarea lui; valoarea NULL indică absența ei.

Valoare returnată:
Funcția returnează valoarea *SUCCESS* în cazul în care operația a reușit, sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obținute sunt:

[*EWRONG_SOURCE_GROUP*]	Grupul sursa nu există
[*EWRONG_DEST_GROUP*]	Grupul destinație nu există
[*ECLIENT_NOT_IN_SOURCE_GROUP*]	Procesul nu aparține grupului sursă
[*EMES_TOO_LONG*]	Dimensiunea mesajului depășește limita maximă
[*ECLIENT_NOT_CONNECTED*]	Procesul nu s-a conectat la sistem

(8) **int ReceiveFromGroup(idGroup** *sgr_id*,**idGroup** *dgr_id*, **int** **pcounter*, **char** **mes*, **int** *timeLimit*, **pf** *pfilter*)

Scop:
\- Permite unui proces sa recepționeze `pcounter` mesaje de la procesele unui grup. Un proces poate recepționa un mesaj numai dacă este membru al grupului la care s-a transmis mesajul receptiv. `Pfilter` este o funcție furnizată de procesul apelant, apelată de TGC pentru fiecare mesaj recepționat. Rolul ei este de a decodifica mesajele. TGC memorează toate mesajele primite pentru un client pînă cînd acesta execută ReceiveFromGroup().

Parametri:
`sgr_id` - identificatorul grupului de la care se dorește recepția de mesaje. Poate fi folosit și identificatorul generic *ANY*, cu sensul că se recepționează mesaje de la orice grup
`dgr_id` - identificatorul grupului către care se transmite mesajul și la care procesul apelant trebuie să fie atașat
`mes` – adresa la care se vor depune mesajele recepționate
`pcounter` – adresa unui întreg, la care, inițial, procesul a depus numărul de mesaje ce se doresc recepționate și la care sistemul returnează numarul de mesaje recepționate.
`pfilter` – pointer la o funcție utilizator apelată de sistemul TGC după recepția fiecărui mesaj, pentru decodificarea lui; valoarea NULL indică absența ei.
`timeLimit` - Poate avea următoarele valori:

- -*INFINITE*, cînd apelantul este pus în așteptare pînă la primirea a *counter* mesaje,
- -o valoare întreagă *n* care semnifică punerea în așteptare a apelantului pentru maximum *n* secunde, dacă în buffer nu există *pcounter* mesaje; după *n* secunde se livrează toate cele *pcounter* mesaje sau cîte s-au găsit în buffer
- -*NOW* care semnifică returnarea imediată a controlului cu livrarea a maximum *pcounter* mesaje.

La returnarea controlului, la `pcounter` se returnează numărul de mesaje recepționate

Valoare returnată:
Funcția returnează valorile:

BUPT

-*SUCCESS*, în cazul în care s-a recepționat cel puțin un mesaj și nu mai sunt alte mesaje în așteptare în buffer,
-*MORE_MES*, în cazul în care s-a recepționat cel puțin un mesaj și mai sunt și alte mesaje în așteptare în buffer,
-*NO_WAIT_MESSAGES* în cazul în care nu s-a recepționat nici un mesaj.
-un cod de eroare în caz contrar. Codurile de eroare care pot fi obținute sunt:

[*EWRONG_SOURCE_GROUP*] Grupul sursa nu există
[*EWRONG_DEST_GROUP*] Grupul destinație nu există
[*ECLIENT_NOT_IN_DEST_GROUP*] Procesul nu aparține grupului destinație
[*ENO_BUF*] Nu există spațiu suficient pentru înregistrarea procesului ca blocat, în așteptare de mesaje.
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(8) `int GetNoOfMessages(idGroup` *`source_gr_id`*`,idGroup` *`dest_gr_id`*`, int` *`*pcounter`*`)`

Scop:
Permite unui proces sa determine la `pcounter` numărul de mesaje, aflate in buffer, în așteptare pentru recepție.

Parametri:
`source_gr_id` - identificatorul grupului de la care se dorește recepția de mesaje. Poate fi folosit și identificatorul generic *ANY*, cu sensul că se recepționează mesaje de la orice grup
`dest_gr_id` - identificatorul grupului către care s-a transmis mesajul și la care procesul apelant trebuie să fie atașat
`pcounter` - adresa unui întreg la care sistemul returnează numărul de mesaje recepționate.

Valoare returnată:
Funcția returnează valoarea *SUCCESS* sau un cod de eroare. Codurile de eroare care pot fi obținute sunt:

[*EWRONG_SOURCE_GROUP*] Grupul sursă nu există
[*EWRONG_DEST_GROUP*] Grupul destinație nu există
[*ECLIENT_NOT_IN_DEST_GROUP*] Procesul nu aparține grupului sursă
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(9) `int InitTGC(char` *`*user`*`,char` *`*autoProto`*`, char` *`*autoString`*`)`

Scop:
Permite conectarea unui proces la sistemul TGC.

Parametri:
`user` - numele utilizatorului
`autoProto` - specifică protocolul de autorizare utilizat de serverul local pentru a determina dacă se permite unui proces să aibă acces la TGC sau nu
`autoString` - șir (parola) specific autorizării

Valoare returnată:
Funcția returnează valoarea *SUCCESS* în cazul în care conectarea a reușit, sau un cod de eroare in caz contrar. Codurile de eroare care pot fi obținute sunt:

[*ENO_AUTHORIZATION*] Conectarea nu este autorizată in urma verificării parolei
[*ETOO_MANY_CLIENTS*] Sunt conectate deja maximum de procese la serverul local
[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

(10) `int CloseTGC(void)`

Permite deconectarea unui proces de la serverul local.

Valoare returnată:
Funcția returnează valoarea *SUCCESS* în cazul în care operația a reușit, sau un cod de eroare in caz contrar:

[*ECLIENT_NOT_CONNECTED*] Procesul nu s-a conectat la sistem

Toate funcțiile descrise mai sus vor fi puse la dispoziția aplicațiilor utilizator într-o bibliotecă de funcții.

BUPT

5.2.2. Varianta semidistribuită

Se propune ca arhitectură a sistemului TGC o arhitectură semidistribuită.

5.2.2.1. Arhitectura generală a sistemului TGC

Întreaga proiectare se bazează pe interacţiunea client-server dintre procese. În fiecare nod TGC există cîte un server local care comunică cu un server global, situat pe un nod fixat în reţea (nodul central). Clienţii comunică cu serverele locale de pe nodurile lor, utilizînd funcţii puse la dispoziţie de TGC prin intermediul unei biblioteci.

Arhitectura sistemului este prezentată în figurile 5.2.2.1.1. şi 5.2.2.1.2.

Această structură distribuită oferă următoarele avantaje:

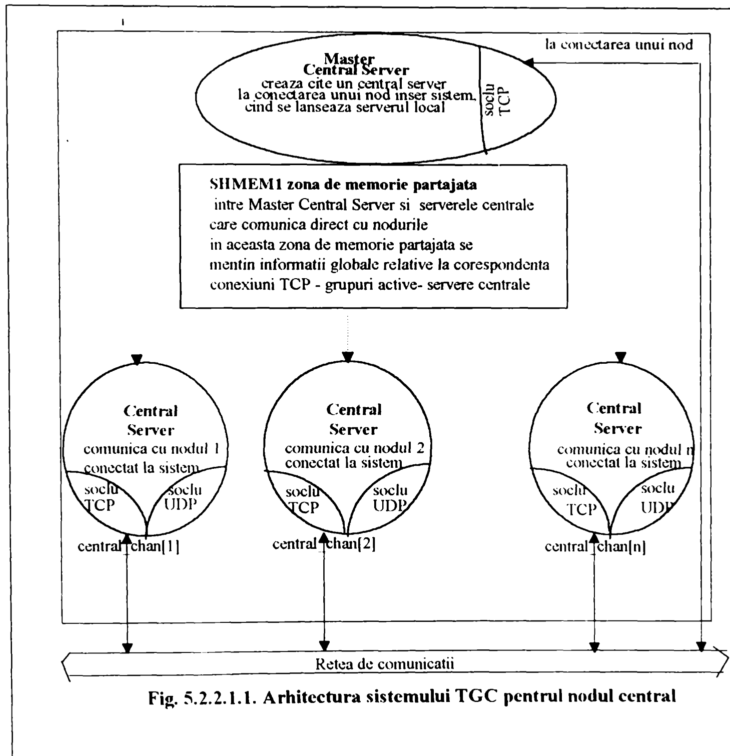


Fig. 5.2.2.1.1. Arhitectura sistemului TGC pentrul nodul central

(1) elimină posibilitatea sufocării unui nod, suprasolicitat într-o variantă centralizată;

(2) asigură, într-un anumit grad, siguranţă în funcţionare: dacă unul dintre nodurile serverelor locale suferă o avarie, atunci serverul global închide conexiunea cu nodul respectiv şi îşi actualizează corespunzător tabelele interne;

(3) în sistemul distribuit, serverul global trebuie să transmită mesaje, numai către serverele locale; acestea, la rîndul lor vor fi cele care transmit aceste mesaje către toate procesele client interne.

Sistemul TGC este compus din următoarele elemente:

(1) *biblioteca de funcţii* pusă la dispoziţie programelor client; fiecare program client trebuie linkeditat cu aceasta bibliotecă. Ea implementează primitivele descrise în subcapitolul 5.2.2.3., pe care un client le poate apela. Aceste primitive sunt transformate în pachete corespunzatoare protocolului de comunicaţie şi trimise serverului local;

(2) *serverul local*, care se execută în fiecare nod TGC memorează o listă a proceselor client locale impreuna cu apartenenţa lor la grupuri. Serverele locale asigură distribuţia mesajelor la procesele de pe acelaşi nod (distribuţie locală). Fiecare client trebuie să se conecteze la serverul său local utilizînd un mecanism IPC (cozi de mesaje, FIFO);

(3) *serverul global* care se execută pe un singur nod al sistemului. Rolul său este de a menţine informaţii globale despre sesiunile multicast (grupurile active şi identitatea nodurilor TGC) şi de a distribui mesajele

BUPT

între noduri. Mesajele către un grup multicast sunt trimise către un server local numai dacă pe acel nod grupul respectiv este activ (dacă există cel puţin un client conectat la acel grup).

Spaţiul numelor alias se memorează în nivelul sistemului global; astfel, serverele locale nu tratează operaţiile relative la alias-uri. Localizarea serverului global este transmisă serverelor locale fie ca variabile de mediu, fie ca parametri în linia de comandă la începutul execuţiei.

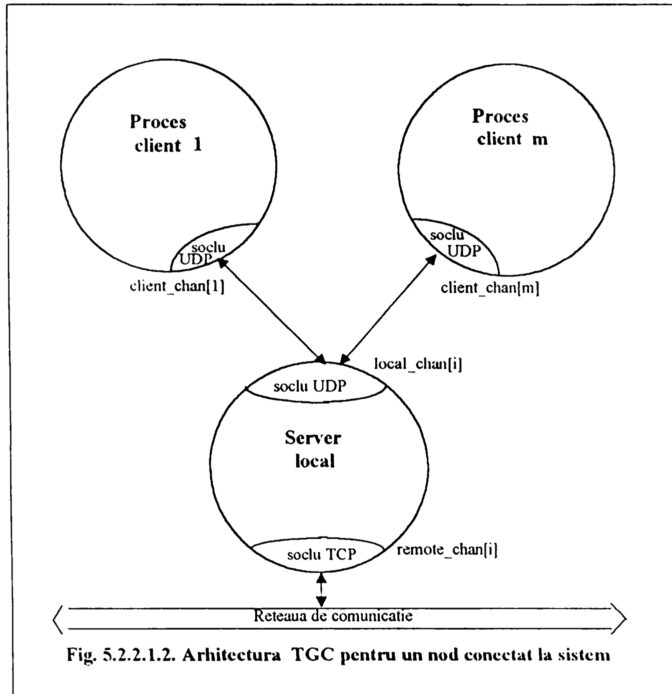


Fig. 5.2.2.1.2. Arhitectura TGC pentru un nod conectat la sistem

5.2.2.2. Protocolul de comunicaţie

TGC foloseşte pentru comunicaţia dintre serverele locale şi clienţi protocolul UDP, fiind adecvat pentru situaţia mai-mulţi-la-unul din acest caz. Comunicaţia în reţea între serverele locale şi serverele centrale se face prin protocolul TCP. Pentru fiecare nod care se conectează in sistem, serverul central master creaza cîte un fiu, un server central, cu rolul de a superviza comunicaţia cu serverul local din nodul respectiv.

O cerere adresată sistemului TGC are următoarea structură:

knd: *kind*. # tipul cererii
client_id: **int** # identificatorul clientului de la care s-a trimis cererea
dest_group_id: **int** # identificatorul grupului de la care s-a trimis cererea
source_group_id: **int** # identificatorul grupului sursa care a trimis mesajul sau cererea
group_name[1:*MAX_LEN_GROUP*]: **char** # nume de grup
len: **int**. # lungimea partii variabile a mesajului
mes[1:len]: **char** # mesajul

Răspunsurile returnate de serverul central au structura următoare:

knd: *kind*. # tipul cererii
error_code: **int** . # codul erorii sau zero pentru succes
client_id: **int** # identificatorul clientului de la care s-a trimis cererea
dest_group_id: **int**. # identificatorul grupului destinatie
source_group_id: **int**. # identificatorul grupului sursa
len: **int**. # lungimea partii variabile a mesajului
mes[1:*len*]: **char** # mesajul

BUPT

5.2.2.3. Descrierea serverelor locale și a serverului central

În cele ce urmează, se prezintă descrierea SR a serverelor locale si serverului central (master si slave). Pentru fiecare nod care se conectează la sistem, serverul master central creaza cîte un proces fiu cu rolul de a superviza comunicația cu acel nod.

Sunt prezentate mai întîi tipurile de mesaje definite pentru TGC si structura informației care circulă pe canalele de comunicație clienți - servere locale și servere locale - servere centrale..

```
type kind = enum (              # tipuri de mesaje care circula in sistem
        INIT_TGC, CLOSE_TGC, CONNECTION_REQ, CREATE_GROUP, JOIN_GROUP,
        GET_ID_GROUP, LEAVE_GROUP, CREATE_ALIAS_GROUP,
        DESTROY_ALIAS_GROUP, SEND_TO_GROUP, RECEIVE_FROM_GROUP_NOW,
        RECEIVE_FROM_GROUP_INFINITE, NO_MORE_RECEIVE_FROM_GROUP,
        SEND_MESSAGE, GET_NO_OF_MESSAGES
        )

type error = enum (       # codificarea raspunsurilor la cererile trimise de clienti
        SUCCESS, MORE_MES, NO_WAIT_MESSAGES, ECLIENT_NOT_CONNECTED,
        ENO_AUTHORIZATION, ETOO_MANY_CLIENTS, ETOO_MANY_CONC_GROUPS,
        EFULL_GROUP, ECLIENT_ALREADY_CONNECTED ,
        ECLIENT_NOT_ IN_DEST_GROUP, EWRONG_SOURCE_GROUP,
        ECLIENT_NOT_ IN_SOURCE_GROUP, EWRONG_DEST_GROUP,
        ENO_BUF, EGROUP_ALREADY_CREATED, ETOO_MANY_GROUPS,
        ETOO_MANY_HOSTS_IN_GROUP, EWRONG_ALIAS, EMES_TO_LONG, EBADF
        )
chan client_chan[1..n, 1..MAX_CLI] (      # canal pentru comunicatie server local → client
        error_code: int ,                 # codul erorii sau zero pentru succes
        dest_group_id: int,               # identificatorul grupului destinatie
        source_group_id: int,             # identificatorul grupului sursa
        len: int,                         # lungimea partii variabile a mesajului
        mes[1:len]: char                  # mesajul
        )
chan local_chan[1..n] (   →     # canal pentru comunicatie client → server local
        knd: kind,              # tipul cererii
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int      # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int    # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                          # lungimea partii variabile a mesajului
        mes[1:len]: char                   # mesajul
        )
chan central_chan[1..n](        # canal pentru comunicatie server local → server central
        knd: kind,              # tipul cererii
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int      # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int    # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                          # lungimea partii variabile a mesajului
        mes[1:len]: char                   # mesajul
        )
chan remote_chan[1..n] (        # canal pentru comunicatie server central → server local
        knd: kind,              # tipul cererii
        error_code: int ,       # codul erorii sau zero pentru succes
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int,     # identificatorul grupului destinatie
        source_group_id: int,   # identificatorul grupului sursa
        len: int,               # lungimea partii variabile a mesajului
        mes[1:len]: char        # mesajul
```

```
                )
chan send_chan[1..n] (          # canal pentru comunicatie server central → server central
                knd: kind,      # tipul cererii, numai SEND_MESSAGE
                error_code: int ,       # codul erorii sau zero pentru succes
                client_id: int  # identificatorul clientului de la care s-a trimis cererea
                dest_group_id: int,     # identificatorul grupului destinatie
                source_group_id: int,   # identificatorul grupului sursa
                len: int,               # lungimea partii variabile a mesajului
                mes[1:len]: char        # mesajul
                )
chan conn_req_chan (            # canal pentru conectarea unui server local la serverul central master
                knd: kind,      # tipul cererii, numai CONNECTION_REQ
                sender: int     # numarul nodului care se conecteaza
                )
chan temp_chan (                # canal pentru conectarea unui client la serverul local
                error_code: int, # codul erorii
                client_id: int  # identificatorul clientului
                ).
```

Serverele locale au următoarea structură:

```
LocalServer[i:1..n]::

const MAX_NAME_LEN = ...                # lungimea maxima a numelor clientilor sau a grupurilor
const MAX_MES_LEN = ...                 # lungimea maxima a unui mesaj
const MAX_CLIENTS_IN_GROUP = ...        #  numarul maxim de clienti dintr-un grup
const MAX_MES_FOR_A_CLIENT = ...        # numarul maxim de mesaje care pot fi buferate pentru un client
const MAX_CONC_GROUPS = ...             # numarul maxim  de grupuri la care poate apartine un client
const MAX_GROUPS = ..                   # numarul maxim de grupuri
const MAX_CLI = MAX_CLIENTS_IN_GROUP * MAX_GROUPS # numarul maxim de clienti
const MAX_PROTO = ... # lungimea maxima a unui sir care indica protocolul folosit pentru autorizare clienti
const MAX_STRING = ... # lungimea maxima a unei parole folosita pentru autorizarea clientilor
const HEADER_LEN = ... # lungimea header-ului mesajelor

type CLIENT_CONN = record of
                (pid: int,                      # identificatorul unui client
                 port: int,                     # identificatorul canalului rezervat dinamic unui client
                 name[1:MAX_NAME_LEN]:char # numele clientului dat la conectare
                )                               #  structura care mentine identitatea clientilor
type MES_TO_GROUP = record of   # structura care mentine informatii  relative la un mesaj receptionat
                (mes[1:MAX_MES_LEN]: char,      # mesaj
                 counter: int,                  # nr proceselor care partajeaza mesajul
                 source_gid: int,               # identificatorul grupului sursa
                 dest_gid: int                  # identificatorul grupului destinatie
                )
type GROUP_OF_CLIENTS = record of               # structura care mentine componenta unui grup
                (gid: int,                      # identificatorul de grup
                 clients_no: int,               # numarul de clienti din acest grup
                 pid_tab[1:MAX_CLIENTS_IN_GROUP]: int           # identificatori clienti
                )
type MES_TO_CLIENT = record of                  # buffer de mesaje pentru un client
                (pid: int,                      # identificatorul clientului
                 pmes[MAX_MES_FOR_A_CLIENT]: pointer to MES_TO_GROUPS # indica mesajul
                )
type WCLIENTS_FOR_GROUP = record of # structura care mentine identificatorii proceselor blocate,
                                # in asteptare de mesaje de la un grup
                (gid: int,      # identificatorul de grup
                wclients_no: int,       # numarul de procese in asteptare de mesaje de  la grupul gid
                wpid_tab[MAX_CLIENTS_IN_GROUP]: int, # identificatorii proceselor in asteptare
```


BUPT

```
            )
type CLIENTS_TO_GROUPS = record of   # structura care mentine grupurile la care apartine un client
            (pid :int                                  # identificatorul clientului
             gid_no: int                               # numarul de grupuri la care apartine un client
             gid_tab[MAX_CONC_GROUPS]: int             # identificatorii grupurilor
            )

var groups_of_clients_tab[1:MAX_GROUPS]: GROUP_OF_CLIENTS # mentine grupurile active din nod
var mes_to_clien_tab[1:MAX_CLI]: MES_TO_CLIENT        # buffere de mesaje pentru clienti
var client_conn_tab[MAX_CLI]: CLIENT_CONN             # mentine clientii activi si conexiunile lor
var wclients_for_group tab[1:MAX_GROUP]:WCLIENT_FOR_GROUP# mentine identitatea clientilor blocati
var client_to_groups_tab[1: MAX_CLI]: CLIENT_TO_GROUPS # mentine grupurile tuturor clientilor
var knd:int, error_code:int,client_id:in, dest_group_id:int, source_group_id:int # folosite pentru citire mesaje
var mes_len: int, mes[1:MAX_MES]: char  # memoreaza lungimea si partea variabila a unui mesaj
var name[1: MAX_NAME_LEN]:char          # folosita pentru numele clientilor si grupurilor (alias-uri)
var auto_proto[1:MAX_PROTO]:char # folosita pentru citirea protocolului utilizat pentru autorizarea clientilor
var auto_string[1:MAX_STRING]: char     # folosita pentru citirea unei parole
var mes_to_groups: MES_TO_GROUP         # folosita la alocarea mesajelor receptionate
var host_name[1:MAX_NAME_LEN]           # folosita la memorarea numelui propriului nod
var pmessage: pointer to MES_TO_GROUP          #variabila de lucru
var message[1:MAX_MES]:char, source_gid: int   # variabile de lucru

  initializari : determinarea numelui propriului nod in host_name si adresei sale
  send conn_req_chan(CONNECTION_REQ, host_name, i) #cerere de conexiune la serverul central
  receive remote_chan[i](knd, error_code, client_id, dest_group_id, source_group_id, len , message)
  if ( knd ≠ CONNECTION_REQ) or (error_code ≠ SUCCESS) → exit fi
do true →
  if not empty(local_chan[i]) →
   receive local_chan[i] (knd, client_id, dest_group_id, source_group_id, name, mes_len , mes)
   if knd = INIT_TGC or client_id ∈ client_conn id →
    if knd = INIT_TGC →
        determina protocolul de autorizare client, auto_proto, si parola, auto_string din mes
        valideaza autorizarea clientului si pozitioneaza error_code
        if error_code≠SUCCESS →
            send temp chan(ENO_AUTHORIZATION,0)
        [] error_code = SUCCESS →
            stabileste un identificator pentru client in client_id,
            ca fiind numarul primei intrari libere din client_conn_tab
            if client_id > MAX_CLIENTS → send temp chan(ETOO MANY CLIENTS, 0)
            [] client_id ≤ MAX_CLIENTS →
                          creaza o intrare clients conn tab[client_id]
                          creaza o intrare mes_to_client tab[client_id]
                          send temp chan (SUCCESS, client id)
            fi
        fi
    [] knd = JOIN GROUP →
        if client to_groups tab[client id].gid no > MAX CONC GROUPS →
            send client chan[client id] (ETOO MANY CONC GROUPS, dest group id, 0, 0, NULL)
        []client_to_groups_tab[client_id].gid_no ≤ MAX CONC_GROUPS
          if dest_group id ∈ group of_clients tab →
           if client_id ∉ group of clients_tab[dest group_id] →
              if group_of_clients_tab[dest_group id].clients_no > MAX CLIENTS IN GROUP →
                     send client_chan[client_id](EFULL GROUP, dest group id, 0, 0, NULL)
              [] group_of_clients_tab[dest_group_id].clients_no ≤ MAX CLIENTS IN GROUP →
                      insereaza client_id in group_of clients_tab[dest_group id]
                      insereaza dest_group_id in clients to group tab
```


BUPT

```
                    send client_chan[client_id](SUCCESS, dest_group_id, 0, 0, NULL)
                fi
            [] client_id ∈ group_of_clients_tab[dest_group_id] →
                send client_chan[client_id](ECLIENT_ALREADY_CONNECTED, dest_group_id, 0, 0, NULL)
            fi
        [] dest_group_id ∈ group_of_clients_tab →
            send central_chan[i](JOIN_GROUP, client_id, dest_group_id, 0, NULL, 0, NULL)
        fi
    fi
[] knd = LEAVE_GROUP →
    if dest_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](WRONG_DEST_GROUP, dest_group_id, 0, 0, NULL)
    [] dest_group_id ∈ group_of_clients_tab →
        if client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
            sterge client_id din group_of_clients_tab[dest_group_id].pid_tab
            fa mesajele i st mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id →
                pmessage := mes_to_clients_tab[client_id].pmes[i]
                mes_to_clients_tab[client_id].pmes[i] = NULL;
                pmessage^.counter :=pmessage^.counter - 1        # un mesaj in asteptare mai putin
                if pmessage^.counter = 0 →  deazaloca structura de la pmessage fi
            fa
        [] client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
            send client_chan[client_id](ECLIENT_NOT_IN_DEST_GROUP, dest_group_id, 0, 0, NULL)
        fi
    fi
[] knd = CREATE_ALIAS_GROUP or knd = DESTROY_ALIAS_GROUP or
                    knd = GET_ID_GROUP or knd = CREATE_GROUP →
    send central_chan[i](knd, client_id, dest_group_id, 0, name, 0, NULL)
[] knd = SEND_TO_GROUP →
    if source_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](EWRONG_SOURCE_GROUP,    # se incearca o emisie dintr-un grup
                    dest_group_id,source_group_id,0,NULL)   # sursa inexistent
    [] source_group_id ∈ group_of_clients_tab →
        if client_id ∉ group_of_clients_tab[source_group_id].pid_tab →
            send client_chan[client_id](ECLIENT_NOT_IN_SOURCE_GROUP,   # se incearca o emisie
                    dest_group_id, 0, 0, NULL)   # in contul unui client neconectat la sursa

        [] client_id ∈ group_of_clients_tab[source_group_id].pid_tab →
            formeaza la message un mesaj din adresa proprie IP, client_conn_tab[client_id].name, mes
            send central_chan[i](SEND_TO_GROUP, client_id, dest_group_id, source_group_id, NULL,
                    mes_len + HEADER_LEN, message)
        fi
    fi
[] knd = RECEIVE_FROM_GROUP_INFINITE →
    if (source_group_id > MAX_GROUPS) →
        send client_chan[client_id](EWRONG_SOURCE_GROUP,   # se incearca receptia de la un grup
                    dest_group_id, source_gid, mes_len,message)  # inexistent
    [] (source_group_id ≤ MAX_GROUPS) →
        if dest_group_id ∉ group_of_clients_tab →
            send client_chan[client_id](EWRONG_DEST_GROUP #se incearca o receptie in contul unui grup
                    dest_group_id, source_group_id,0, NULL)# care nu exista
        [] dest_group_id ∈ group_of_clients_tab →
            if client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
            send client_chan[client_id](ECLIENT_NOT_IN_DEST_GROUP,   # se incearca o receptie in
```


BUPT

```
                    dest_group_id, 0, 0, NULL)   # in contul unui client neconectat la grupul destinatie
        [] client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
          if exista mesaje i in mes_to_client_tab[client_id] and
                    mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                    (source_group_id = ANY or
                    ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
             extrage primul mesaj i  in message si calculeaza lungimea sa in mes_len
             pmessage := mes_to_clients_tab[client_id].pmes[i]
             mes_to_clients_tab[client_id].pmes[i] = NULL;
             source_gid = pmessage^.source_gid
             pmessage^.counter :=pmessage^.counter - 1      # un mesaj in asteptare mai putin
             if pmessage^.counter = 0 →  deazaloca structura de la pmessage fi
             if (mai exista mesaje i in mes_to_client_tab[client_id]) and
                      mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                      ((source_group_id = ANY ) or
                      ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
                send client_chan[client_id](MORE_MES, dest_group_id, source_gid,
                      mes_len, message) # se receptioneaza mesajul si o indicatie ca mai exista mesaje
              [] nu mai exista mesaje in asteptare de la source_group_id catre dest_group_id →
                send client_chan[client_id](SUCCESS, dest_group_id, source_gid, mes_len,message)
                      #se receptioneaza mesajul si o indicatie ca nu mai exista mesaje
              fi
          []nu exista mesaje in asteptare de la source_group_id catre dest_group_id pentru client_id →
             if wclients_for_groups[source_group_id].wclients_no>MAX_CLIENTS_IN_GROUP →
                send client_chan[client_id](ENO_BUF, dest_group_id, source_gid, mes_len,message)
             [] wclients_for_groups[source_group_id].wclients_no ≤ MAX_CLIENTS_IN_GROUP →
                insereaza client_id in wclients_for_groups[source_group_id].wpid.tab
                si actualizeaza wclients_for_groups[source_group_id].wclients_no
                if source_group_id ∉ group_of_clients_tab →
                send central_chan[i](RECEIVE_FROM_GROUP_INFINITE, client_id,
                                              dest_group_id,source_group_id, NULL,0, NULL)
                                              # pentru verificare source_group_id
                fi
             fi
          fi
        fi
      fi
    fi
  [] knd = NO_MORE_RECEIVE_FROM_GROUP →
      sterge client_id wclients_for_groups[source_group_id].wpid.tab
      si actualizeaza wclients_for_groups[source_group_id].wclients_no
  [] knd = RECEIVE_FROM_GROUP_NOW →
    if (source_group_id > MAX_GROUPS) →
      send client_chan[client_id](EWRONG_SOURCE_GROUP,    # se incearca receptia de la un grup
                    dest_group_id, source_gid, mes_len,message)  # inexistent
    [] (source_group_id ≤ MAX_GROUPS) →
      if dest_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](EWRONG_DEST_GROUP  #se incearca o receptie in contul unui grup
                    dest_group_id, source_group_id,0, NULL)# care nu exista
      [] dest_group_id ∈ group_of_clients_tab →
        if client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
          send client_chan[client_id](ECLIENT_NOT_IN_DEST_GROUP,   # se incearca o receptie in
                    dest_group_id, 0, 0, NULL)   # in contul unui client neconectat la grupul destinatie
        [] client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
```

```
if exista mesaje i in mes_to_client_tab[client_id] and
        mes_to_clients_tab[client_id].pmes[i]^.dest_gid   dest_group_id and
        (source_group_id = ANY or
        ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
    extrage primul mesaj i  in message si calculeaza lungimea sa in mes_len
    pmessage := mes_to_clients_tab[client_id].pmes[i]
    mes_to_clients_tab[client_id].pmes[i] = NULL;
    source_gid = pmessage^.source_gid
    pmessage^.counter :=pmessage^.counter - 1      # un mesaj in asteptare mai putin
    if pmessage^.counter = 0 →   deazaloca structura de la pmessage  fi
    if (mai exista mesaje i in mes_to_client_tab[client_id]) and
            mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
            ((source_group_id = ANY) or
            ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
        send client_chan[client_id](MORE_MES, dest_group_id, source_gid,
            mes_len, message) # se receptioneaza mesajul si o indicatie ca mai exista mesaje
    [] nu mai exista mesaje in asteptare de la source_group_id catre dest_group_id →
        send client_chan[client_id](SUCCESS, dest_group_id, source_gid, mes_len,message)
            #se receptioneaza mesajul si o indicatie ca nu  mai exista mesaje
    fi
[] nu exista mesaje in asteptare de la  source_group_id catre dest_group_id →
    send client_chan[client_id](NO_WAIT_MESSAGES, dest_group_id, source_gid,
                                0,NULL) #nu exista mesaje pentru receptie
        fi
      fi
    fi
  fi
[] knd = GET_NO_OF_MESSAGES →
  if (source_group_id > MAX_GROUPS) →
      send  client_chan[client_id](EWRONG_SOURCE_GROUP,    # se incearca receptia de la un grup
                dest_group_id, source_gid, mes_len,message)  # inexistent
  [] (source_group_id ≤ MAX_GROUPS) →
      if dest_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](EWRONG_DEST_GROUP  #se incearca o receptie in contul unui grup
                dest_group_id, source_group_id,0, NULL)# care nu exista
      [] dest_group_id ∈ group_of_clients_tab →
        if client_id  ∉  group_of_clients_tab[dest_group_id].pid_tab →
          send client_chan[client_id](ECLIENT_NOT_IN_DEST_GROUP,   # se incearca o receptie in
                dest_group_id, 0, 0, NULL)   # in contul unui client neconectat la grupul destinatie
        [] client_id  ∈  group_of_clients_tab[dest_group_id].pid_tab →
          contorizeaza in k mesajele i cu
                mes_to_client[client_id].pmes[i]^.source_gid = source_group_id s
                mes_to_client[client_id].pmes[i]^.dest_gid = dest_group_id
                send client_chan[client_id](SUCCESS, dest_group_id, source_group_id, k, NULL)
        fi
      fi
  fi
[] knd = CLOSE_TGC →
    sterge intrarea client_id din client_conn_tab[client_id]
    fa toate grupurile la care este conectat client_id, client_to_groups[client_id].gid.tab[i] →
      source_gid := client_to_groups[client_id].gid.tab[i]
      sterge client_id din group_of_clients_tab[source_gid].gid_tab[i]
          fa mesajele i st mes_to_clients_tab[client_id].pmes[i]^.dest_gid = source_gid →
            pmessage := mes_to_clients_tab[client_id].pmes[i]
            mes_to_clients_tab[client_id].pmes[i] = NULL;
```


BUPT

```
                pmessage^.counter :=pmessage^.counter - 1          # un mesaj in asteptare mai putin
                if pmessage^.counter = 0 → deazaloca structura de la pmessage fi
              fa
          af
       fi
     fi
    fi
    if not empty(remote_chan[i]) →
       receive remote_chan[i] (knd, error_code, client_id, dest_group_id, source_group_id, mes_len, message)
       if knd = CREATE_GROUP, JOIN_GROUP →
           if error_code = SUCCESS →
               adauga o intrare in group_of_clients[dest_group_id] care contine client_id
               insereaza dest_group_id in clients_to_group_tab[client_id]
           [] error_code ≠ SUCCESS →
               send client_chan[client_id](error_code, dest_group_id,0,0, NULL)
           fi
       [] knd = CREATE_ALIAS_GROUP or knd = DESTROY_ALIAS_GROUP
                                    or knd = GET_ID_GROUP or knd = SEND_TO_GROUP →
               send client_chan[client_id](error_code, dest_group_id,0,0, NULL)
       [] knd = RECEIVE_FROM_GROUP_INFINITE →
           if error_code = SUCCESS →
               sterge client_id din wclients_for_groups[source_group_id].wpid.tab
               send client_chan[client_id](error_code, dest_group_id, source_group_id,0, NULL)
           fi
       [] knd = SEND_MESSAGE →
           if exista client_id astfel incit (client_id ∈ dest_group_id) and
              (client_id ∉ wclients_for_group[source_group_id].wpid_tab) →
              se aloca o zona de memorie pentru o structura MES_TO_GROUP. pmessage
               pmessage^.mes := mes
               pmessage^.source_gid := source_group_id
               pmessage^.dest_gid := dest_group_id
              se contorizeaza in k toti membrii grupului dest_group care nu sunt in
                                                   wclients_for_group[source_group_id].wpid_tab
               pmessage^.counter := k               fi
           fa client_id ∈ dest_group_id st client_id ∉ wclients_for_group[dest_group_id].wpid_tab
               insereaza pmessage in mes_to_client_tab[client_id].pmes
           fa
           fa (client_id ∈ dest_group_id) st (client_id ∈ wclients_for_group[source_group_id].wpid_tab)
               send client_chan[client_id](SUCCESS, dest_group_id, source_group_id, mes_len,mes)
                                  #se receptioneaza mesajul si o indicatie ca nu mai exista mesaje
               sterge client_id din wclients_for_groups[source_group_id].wpid.tab
            af
         fi
     fi
    od
```

Serverele centrale pot fi descrise astfel:

```
const MAX_HOSTS_FOR_A_GROUP          # numarul maxim de noduri pe care poate fi deschis un grup
const MAX_HOSTS = ...                # numarul maxim de noduri care se pot conecta la sistem
const MAX_NAME_LEN = ...                     # lungimea maxima a numelor clientilor sau a grupurilor
const MAX_ALIASES_FOR_A_GROUP = ...          # numarul maxim de pseudonime pentru un grup
const MAX_HOSTS_NAME = ...           # lungimea maxima a numelui unui nod
const MAX_ADDRESSE_LEN = ...         # lungimea maxima a adresei unui nod
```


BUPT

```
type HOSTS_FOR_GROUP = record of    # structura care mentine nodurile pe care s-a deschis grupul gid
        (gid: int,                  # identificatorul de grup
         hosts_no: int,             # numarul de noduri pe care exista membri ai grupului gid
         port_no[MAX_HOSTS_FOR_A_GROUP]: int # porturile canalelor proceselor CentralServer
        )                           # care controleaza comunicatia cu nodurile grupului
type GROUP_ALIASES = record of      # structura care mentine pseudonimele grupului gid
        (gid: int,
         alias_no: int,             # numarul de pseudonime acordate grupului gid
         alias_tab[MAX_ALIASES_FOR_A_GROUP][MAX_NAME_LEN]: char # pseudonime
        )
type HOSTS = record of              # structura care mentine informatii despre noduri
        (name[MAX_HOSTS_NAME]: char,
         addresse[MAX_ADDRESSE_LEN]: char,
         port: int;                 # porturile canalelor proceselor CentralServer
        )                           # care controleaza comunicatia cu nodurile grupului


var hosts_for_group_tab[MAX_GROUPS]: HOSTS_FOR_GROUP   #mentine repartitia grupurilor pe noduri
var group_aliases_tab[MAX_GROUPS]: GROUP_ALIASES       # mentine pseudonimele grupului
var hosts_tab[MAX_HOSTS]: HOSTS                        # mentine informatii despre noduri
```

MasterCentralServer::

```
var knd: kind, host_name[MAX_NAME_LEN]: char # pentru citirea mesajelor
 initializari
 do true →
    receive conn_req_chan(knd, host_name, i) # cerere de conexiune primita de la un server local i
    if ( knd ≠ CONNECTION_REQ)→
        send remote_chan[i](CONNECTION_REQ, BAD_REQUEST, 0, 0, 0, 0, NULL )
    [] if nu exista o intrare liberea in hosts_tab →
        send remote_chan[i](CONNECTION_REQ, TOO_MANY_HOSTS, 0, 0, 0, 0, NULL )
    [] knd = CONNECTION_REQ and exista o intrare liberea in hosts_tab →
        creaza si completeaza o intrare i in hosts_tab
        creaza un proces CentralServer si transmite ca parametru hosts_tab[i]
 od
```

CentralServer::

```
var knd: int, error_code: int, client_id: int                        # folosite pentru mesaje
var dest_group_id: int, source_group_id: int, gid: int              # folosite pentru mesaje
var name[1:MAX_NAME_LEN]:char,mes_len: it, message[1:MAX_MES]): char #variabile de lucru
do true →
  if not empty(central_chan[i]) →
    receive central_chan[i]( knd, client_id, dest_group_id, source_group_id, name, mes_len, message)
      if knd   CREATE_GROUP →
        if exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            send remote_chan[i](knd,EGROUP_ALREADY_CREATED, client_id,0, 0, 0, NULL)
        [] nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            if nu exista o intrare o intrare libera in hosts_for_group_tab →
                send remote_chan[i](knd,ETOO_MANY_GROUPS, client_id, 0, 0, 0, NULL)
            [] exista o intrare o intrare libera in hosts_for_group_tab →
                creaza si completeaza o intrare gid in hosts_to_groups_tab si in group_aliases_tab
                send remote_chan[i] (CREATE_GROUP, SUCCESS, client_id, gid, 0, 0 NULL)
            fi
        fi
      [] knd   GET_ID_GROUP →
        if nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
```


BUPT

```
            send remote_chan[i] (GET_ID_GROUP, EWRONG_ALIAS, client_id, 0, 0, 0, NULL)
        [] exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            send remote_chan[i] (GET_ID_GROUP, SUCCESS, client_id, gid, 0, 0, NULL)
        fi
    [] knd = CREATE_ALIAS_GROUP →
        if dest_group_id ∉ hosts_for_group_tab →
            send remote_chan[i] (knd, EWRONG_DEST_GROUP, client_id, dest_group_id, 0, 0, NULL)
        [] dest_group_id ∈ hosts_for_group_tab →
            if exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
                send remote_chan[i] (CREATE_ALIAS_GROUP, EWRONG_ALIAS, client_id, 0,0,0,NULL)
            [] nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
                if group_aliases_tab[dest_group_id].alias_no > MAX_ALIASES_FOR_A_GROUP →
                    send remote_chan[i] (knd, ETOO_MANY_ALIASES,client_id,dest_group_id ,0,0,NULL)
                [] group_aliases_tab[dest_group_id].alias_no ≤ MAX_ALIASES_FOR_A_GROUP →
                    adauga nume in group_aliases_tab[dest_group_tab]
                    send remote_chan[i] (CREATE_ALIAS_GROUP, SUCCESS, client_id, gid, 0, 0, NULL)
                fi
            fi
        fi
    [] knd = DESTROY_ALIAS_GROUP →
        if dest_group_id ∉ hosts_for_group_tab →
            send remote_chan[i] (knd, EWRONG_DEST_GROUP, client_id,dest_group_id, 0, 0, NULL)
        [] dest_group_id ∈ hosts_for_group_tab →
            if name ∉ group_aliases_tab[dest_group_id]→
                send remote_chan[i] (DESTROY_ALIAS_GROUP, EWRONG_ALIAS, client_id, 0,0,0,NULL)
            [] name ∈ group_aliases_tab[dest_group_id]→
                sterge nume din group_aliases_tab[dest_group_tab]
                send remote_chan[i] (DESTROY_ALIAS_GROUP, SUCCESS, client_id, 0,0,0,NULL)
            fi
        fi
    [] knd = JOIN_GROUP →
        if dest_group_id ∉ hosts_for_group_tab →
            send remote_chan[i] (knd, EWRONG_DEST_GROUP, client_id, dest_group_id, 0, 0, NULL)
        [] dest_group_id ∈ hosts_for_group_tab →
            if hosts_for_group_tab[dest_group_id].host_no   MAX_HOSTS_FOR_A_GROUP →
                if portul nodului apelant nu exista in hosts_for_group_tab[dest_group_id].port_no→
                adauga portul nodului in intrarea grupului din hosts_for_group_tab[dest_group_id].port_no
                si actualizeaza hosts_for_group_tab[dest_group_id].host_no
                fi
                send remote_chan[i] (JOIN_GROUP, SUCCESS, client_id, dest_group_id, 0, 0, NULL)
            [] hosts_for_group_tab[dest_group_id].host_no ≥ MAX_HOSTS_FOR_A_GROUP →
                send remote_chan[i] (JOIN_GROUP, ETOO_MANY_HOSTS_IN_GROUP,
                                client_id, dest_group_id, 0, 0, NULL)
            fi
        fi
    [] knd   LEAVE_GROUP →
        if dest_group_id ∈ hosts_for_group_tab →
            sterge din hosts_for_group_tab[dest_group_id].port_no portul nodului
            si actualizeaza hosts_for_group_tab[dest_group_id].host_no
            if hosts_for_group_tab[dest_group_id].host_no = 0 →
                sterge intrarile hosts_for_group_tab[dest_group_id] si group_aliases_tab[dest_group_id]
            fi
        fi
        send remote_chan[i] (LEAVE_GROUP, SUCCESS, client_id, dest_group_id, 0, 0, NULL)
    [] knd = RECEIVE_FROM_GROUP_INFINITE →
```


BUPT

```
            if source_group_id ∉ hosts_for_group_tab →
                send remote_chan[i] (knd, EWRONG_SOURCE_GROUP, client_id, dest_group_id,
                                      source_group_id, 0, NULL)          # grup sursa inexistent
            || source_group_id ∈ hosts_for_group_tab →
                send remote_chan[i] (knd, SUCCESS, client_id, dest_group_id, source_group_id, 0, NULL)
            fi
        || knd = SEND_TO_GROUP →
            if dest_group_id ∉ hosts_for_group_tab →
                send remote_chan[i] (knd, EWRONG_DEST_GROUP, client_id, dest_group_id, 0, 0, NULL)
            || dest_group_id ∈ hosts_for_group_tab →
                send remote_chan[i] (knd, SUCCESS, client_id, dest_group_id, 0, 0, NULL)
                fa j = 1 to hosts_for_group_tab[dest_group_id].host_no →
                    send send_chan[hosts_for_group_tab[dest_group_id].port_no[j]] (SEND_MESSAGE,
                          SUCCESS, client_id, dest_group_id, source_group_id, mes_len, message)
                af
            fi
        fi
    fi
    if not empty(local_chan[i]) →
        receive send_chan[i]( knd,error_code,client_id,dest_group_id,source_group_id, name, mes_len, message)
        if knd = SEND_MESSAGE →
            send remote_chan[i]( knd, SUCCESS, client_id, dest_group_id, source_group_id, mes_len, message)
        fi
    fi
od
```

Functiile puse la dispozitia clientilor pentru aplicatii pot fi descrise astfel:

```
const MAX_NAME_LEN =                         # lungimea maxima a numelor clientilor sau a grupurilor
const MAX_PROTO =         # lungimea maxima a unui sir care indica protocolul folosit pentru autorizare clienti
const MAX_STRING =        # lungimea maxima a unei parole folosita pentru autorizarea clientilor
const HEADER_LEN =        # lungimea header-ului mesajului
var client_id: int :=0    # identificatorul clientului returnat de serverul local la conectarea clientului
                          # trebuie transmis in continuare in corpul  fiecarei cereri
var error_code: int, dest_group_id: int, source_group_id: int          # folosite pentru  mesaje
var mes_len: it, message[1:MAX_MES]): char                              #variabile de lucru

InitTGC(user[MAX_NAME_LEN]: char, auto_proto[MAX_PROTO]: char, passw[MAX_STRING]: char)::
        if user = NULL → return EBADF fi
        if client_id ≠ 0 →
                send local_chan[i]((CLOSE_TGC, client_id, 0, 0, NULL, 0, NULL)
                client_id := 0
        fi
        send local_chan[i]((INIT_TGC, 0, 0, 0, user, length(auto_proto + passwd), auto_proto + passwd)
        receive temp_chan (error_code, client_id)
        return error_code

CreateGroup(var pgroup: int, name[MAX_NAME_LEN]: char)::
        if client_id = 0 → return ECLIENT_NOT_CONNECTED
        if name = NULL → return EBADF fi
        send local_chan[i]((CREATE_GROUP, client_id, 0, 0, name,0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        if error_code ≠ SUCCESS  pgroup := dest_group_id
        return error_code
```


BUPT

```
JoinGroup(group_id: int)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        send local_chan[i](JOIN_GROUP, client_id, group_id, 0, NULL, 0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        return error_code

LeaveGroup(group_id: int)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        send local_chan[i](LEAVE_GROUP, client_id, group_id, 0, NULL, 0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        return error_code

CreateAliasGroup(group: int, alias[MAX_NAME_LEN]: char)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        if alias = NULL  → return EBADF fi
        send local_chan[i](CREATE_ALIAS_GROUP, client_id, group_id, 0, name,0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        return error_code

DestroyAliasGroup(group: int, alias[MAX_NAME_LEN]: char)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        if alias = NULL  → return EBADF fi
        send local_chan[i](DESTROY_ALIAS_GROUP, client_id, group_id, 0, name,0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        return error_code

GetIdGroup(var pgroup: int, alias[MAX_NAME_LEN]: char)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        if alias = NULL  → return EBADF fi
        send local_chan[i](GET_ID_GROUP, client_id, 0, 0, name,0, NULL)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        if error_code ≠ SUCCESS   pgroup := dest_group_id
        return error_code

SendToGroup(dest_group_id: int, source_group_id: int, mes_len: int, message[MAX_MES]: char,
                                        counter: int, pfilter: pointer to function)::
        if client_id = 0  → return ECLIENT_NOT_CONNECTED
        if counter > MAX_MES → return  EMES_TO_LONG
        pfilter(message)                    # se codifica mesajul
        mes_len := counter
        send local_chan[i](SEND_TO_GROUP,client_id,dest_group_id,source_group_id, mes_len, message)
        receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
        return error_code

ReceiveFromGroup(source_group_id: int, dest_group_id: int,  var pcounter: int, var pmes[MAX_MES]:
                                        char,  time_limit: int, pfilter: pointer to function)::
     if client_id = 0  → return ECLIENT_NOT_CONNECTED
     if time_limit = NOW → mode = RECEIVE_FROM_GROUP_NOW
     [] time_limit = INFINITE → mode = RECEIVE_FROM_GROUP_INFINITE
     fi
     if  time_limit = NOW or time_limit = INFINITE →
        i := 0; error_code := MORE_MES
        do (i< pcounter)and(error_code=MORE_MES or (error_code=SUCCESS and time_limit=INFINITE))
            send local_chan[i](mode, client_id, dest_group_id, source_group_id, NULL, 0, NULL)
```


BUPT

```
                receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
                if (error_code ≠ MORE_MES) or (error_code ≠ SUCCESS) →
                        pcounter := i; return error_code
                fi
                pfilter(message + HEADER_LEN)
                pmes := pmes + mes_len; memoreaza message la pmes
                i := i + 1
          od
    [] time_limit ≠ NOW and time_limit ≠ INFINITE →
          i := 0; error_code := MORE_MES
          do (i< pcounter) and (error_code=MORE_MES or (error_code=SUCCESS )
                send local_chan[i](RECEIVE_FROM_GROUP_INFINITE, client_id, dest_group_id,
                                                source_group_id, NULL, 0, NULL)
                programeaza un semnal ALARM pentru n secunde, a carui rutina seteaza pe error_code = ALARM
                receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)

                if error_code ≠ MORE_MES or error_code ≠ SUCCESS →
                        pcounter := i; return error_code
                fi
                pfilter(message + HEADER_LEN)
                i := i + 1
          od
    fi
    pcounter := i;
    return error_code

GetNoOfMessages(source_group_id: int, dest_group_id: int, var pcounter: int)::
    if client_id = 0  → return ECLIENT_NOT_CONNECTED
    send local_chan[i](GET_NO_OF_MESSAGES, client_id, dest_group_id, source_group_id,NULL,0,NULL)
    receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
    if  error_code ≠ SUCCESS →  pcounter := 0; return error_code
    [] pcounter : = mes_len; return SUCCESS
    fi

CloseTGC()::
    if client_id = 0  → return ECLIENT_NOT_CONNECTED
    send local_chan[i](CLOSE_TGC, 0, 0, 0, NULL, 0, NULL)
    receive client_chan[client_id] (error_code, dest_group_id, source_group_id, mes_len, message)
    return error_code
```

BUPT

În descrierea de mai sus, nu a fost prezentată soluţia adoptată pentru respectarea atomicităţii în transmisia mesajelor. Deoarece protocolul folosit in reţeaua de comunicaţii este protocolul TCP, singura prevedere ce trebuie luată este cea împotriva situaţiei de "buffere ocupate"; aceasta înseamnă că, de fapt, trebuie asigurat controlul fluxului mesajelor la nivelul aplicaţiei. Mai precis, controlul fluxului mesajelor trebuie asigurat între serverul central şi serverele locale, la execuţia unei operaţii *SEND_MESSAGE*. Soluţia adoptată este următoarea: cînd, în urma primirii unui mesaj, un server local detectează că se ocupă toate bufferele sale şi, în viitor, nu mai poate astfel primi nici un mesaj pentru un anumit grup, deoarece nu va mai avea unde să-l memoreze, informează serverul central despre acest lucru, transmiţîndu-i un mesaj *FULL_BUFFERS*. Serverul central menţine în tabloul *hosts_for_groups_tab[MAX_GROUPS]*, de tipul *HOSTS_FOR_GROUP* încă un cîmp, care arată dacă umplerea bufferelor pentru un nod este iminentă. Dacă, atunci cînd trebuie trimis un mesaj la grupul respectiv, se gaseşte pentru un nod în acest cîmp valoarea 1, serverul global returnează eroarea *ENO_BUFS*.

Tabloul *hosts_for_groups_tab[MAX_GROUPS]* se actualizează atunci, cînd, în nodul în cauză, procesele client recepţionează mesaje de la acel grup, eliberînd astfel buffere; în această situaţie, serverul local transmite serverului global mesajul *FREE_BUFFER*.

5.2.3. Alte variante

Varianta prezentată la 5.2.2. se poate caracteriza ca o variantă semidistribuită: există cîte un server local în fiecare nod şi un server global unic, cu rol de proces coordonator, situat pe un nod fixat în reţea.

În această secţiune vor fi prezentate şi alte soluţii de proiectare a unui toolkit pentru comunicaţii de grup, menţinînd ca valabile următoarele cerinţe de proiectare: (1) sistemul TGC trebuie să se execute în mod utilizator. Singura excepţie poate fi legată de faptul că modul broadcast în Unix poate fi o operaţie privilegiată. Este preferabil însă ca sistemul să se execute în mod utilizator, putînd fi astfel startat şi de utilizatori obisnuiţi. Totusi, unele componente, (eventual cele legate de stabilirea de parole pentru clienţi), dacă este necesar, pot să fie prevăzute cu unele restricţii de execuţie. (2) Protocolul de comunicaţie trebuie să fie independente de hardware.

5.2.3.1. Variante complet distribuite

O variantă complet distribuită prezintă faţă de varianta semidistribuită avantajul eliminării serverului coordonator, dar şi dezavantajul creşterii încărcării reţelei. Există două soluţii: folosirea modului de lucru broadcast, prin utilizarea unui protocol adecvat, şi folosirea unui protocol punct-la-punct.

Arhitectura generală a sistemului, în ambele cazuri, va fi aceeaşi; va exista cîte un server general în fiecare nod. Acesta va menţine aceleaşi structuri de date cu cele ale unui server local din varianta semidistribuită, dar va îngloba şi structurile de date ale serverului central din varianta semidistribuită.

Prin urmare, va conţine atît structurile de date necesare menţinerii evidenţei ataşării clienţilor la grupuri dar şi numele alias asociate tuturor grupurilor. În ceea ce priveşte structurile de date care menţin legatura dintre grupuri şi nodurile pentru care există clienţi ataşaţi la aceste grupuri, ele sunt necesare numai în varianta în care nu se utilizează transmisie broadcast. Va fi necesar însă, în ambele variante introducerea mărcilor de timp în corpul mesajelor, pentru a ordona transmisiile în sistem.

Pentru tratarea transmisiei şi recepţionării de mesaje concurentă cu execuţia operaţiilor şi pentru modularizarea implementării se separă protocolul de comunicaţie într-un proces separat, care livrează cererile de operaţii confirmate unui proces sever, ce tratează cererile. Arhitectura generală a sistemului este prezentată în fig. 5.2.3.1.

A. Varianta în care se foloseste transmisia broadcast

Protocolul folosit va fi protocolul UDP.

Mecanismul confirmărilor va fi construit asemănător cu cel prezentat la 4.3.4.1. în cadrul implementării semafoarelor distribuite. Vor fi tratate de fiecare nod numai mesajele *complet confirmate*. Dacă o cerere nu este complet confirmată dar, în schimb, este urmată de una sau mai multe cereri complet confirmate, după cîteva mesaje de probă cu nodul care nu a confirmat cererea (neconfirmate şi ele) nodul respectiv se deconecteaza din sistem şi nu va mai fi luat în considerare pentru contorizarea confirmărilor decît după o nouă cerere de conectare a sa.

Nu toate cererile de operaţii trebuie transmise în reţea. Operaţiile care nu trebuie transmise broadcast în reţea sunt: *InitTCG(), GetIdGroup(), JoinGroup(), ReceiveFromGroup(), GetNoOfMessages(), Close TCG()*; în plus, operaţia *LeaveGroup()* nu trebuie transmisă în reţea decît dacă genereazā, în urma execuţiei ei, ştergerea unui grup; evident, toate operaţiile sunt transmise în reţea numai dacă nu generează, în urma execuţiei, un cod de eroare de tipul grup inexistent, client neataşat la grup, format eronat al parametrului etc.

Structura unui pachet care transmite o operaţie în reţea va fi:

BUPT

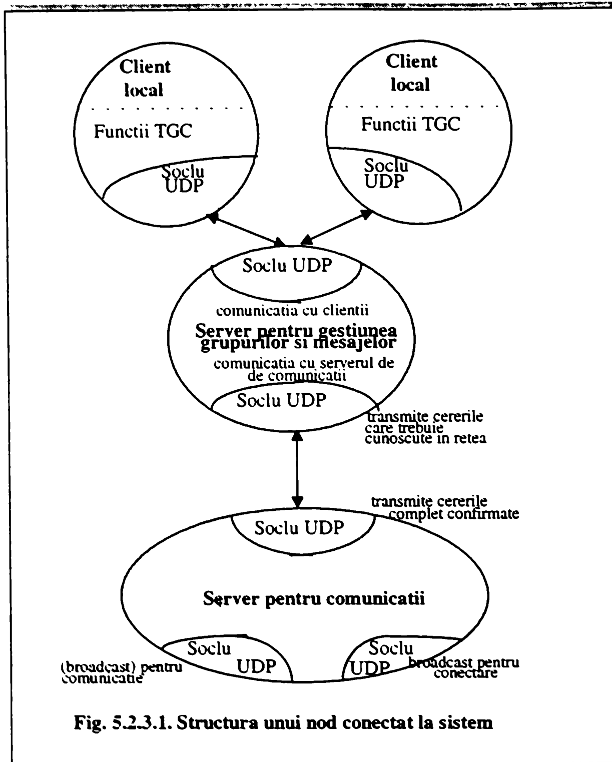


Fig. 5.2.3.1. Structura unui nod conectat la sistem

```
{
    /* toate pachetele au un header de identificare */
    LENGTHS kind ;                  /* tipul cererii */
    ADDRESS sender;                 /* identificatorul emitatorului */
    TS-TYPE time-stamp ;            /* marca de timp */
    LENGTHS duplicate;              /* numarul de retransmisie al mesajului */
    /* partea mesajui primita de la serverul pentru gestiunea grupurilor */
    MESSAGE variable-part ;
}
```

Fiecare pachet are un header format din: (1) o zonă de dimensiune constantă, obligatorie pentru toate tipurile de cereri, care conține un octet pentru identificarea tipului cererii, identificatorul emițătorului și marca de timp și (2) mesajul propriu zis care conține și o zona de dimensiune variabilă (în cazul operației *SendToGroup()* conține mesajul de transmis iar în cazul operațiilor *CreateGroup()*, *CreateAliasGroup()*, *DestroyAliasGroup()* numele de grup implicat în operație).

O atenție specială trebuie acordată operației *CreateGroup()*, care trebuie executată în regim de excludere reciprocă, pentru ca fiecare nod să poată alege un identificator de grup unic la crearea unui grup nou.

Atomicitatea transmisiilor se asigură prin meacanismul special al confirmărilor. Se presupune că mesajele și confirmările primite de un nod sunt identice cu cele primite de toate nodurile. Mesajele nu se pot pierde decît în patru situații: cînd un nod are bufferele ocupate, cînd un nod nu mai răspunde la nici un mesaj, cînd se constată lipsa unui mesaj sau a unei confirmări (în toate nodurile se va observa același lucru). Prima situație se poate rezolva transmițînd totuși un mesaj de confirmare dar cu un conținut care să indice eroarea *ENO_BUFS*. Cea de-a doua situație poate fi detectată ușor, deoarece, dacă un nod nu mai este activ în sistem, toate mesajele următoare vor fi neconfirmate complet. În acest caz, se deconectează automat nodul respectiv din sistem. Pentru ultimele două situații, mesajele neconfirmate complet, urmate de altele confirmate complet sunt ignorate de toate nodurile. Pentru fiecare mesaj trimis în sistem, funcțiile TGC așteaptă să recepționeze un cod de eroare care arată modul în care s-a desfășurat execuția: cu succes sau cu o eroare. În ultimul caz, se reîncearcă transmisia de cîteva ori înainte de a se un răspuns utilizatorului.

Descrierea serverului de comunicații în această variantă este prezentată la 5.2.4.1. iar a managerului de grupuri în anexa A.

BUPT

B. Varianta în care nu se foloseste transmisie broadcast

Protocolul de bază folosit în comunicaţie va fi protocolul TCP; s-a preferat acest protocol deoarece rezolvă automat problemele de retransmisie în cazul unor erori pe liniile de comunicaţie, oferind siguranţă în funcţionare. Se poate folosi însă şi protocolul UDP, dacă se completează cu un mecanism de confirmări.

Structura pachetelor vehiculate în reţea este aceeaşi cu cea prezentată la punctul A. Spre deosebire de varianta A, fiecare nod va trebui să menţină structuri de date referitoare la dispunerea grupurilor pe noduri în sistem.

Atomicitatea transmisiilor se rezolvă printr-un mecanism asemanator protocolului *two-phase commit* (vezi capitolul 7). Acest protocol presupune că orice mesaj se transmite în reţea în două faze:
(1) În prima fază, *faza de pregătire,* se ***încearcă*** dacă transmisia poate reuşi, transmiţîndu-se mesajul la toate nodurile receptoare: acestea nu transmit mesaje la clienţi, ci confirmă doar faptul că pot primit mesajul;
(2) În cea de a doua fază, *faza de **realizare,*** nodul emiţător colectează toate mesajele de confirmare şi, în momentul în care s-au primit toate confirmările, transmite mesajul ***commit*** către toate nodurile receptoare, care, numai în acest moment, iau în considerare mesajul primit. Dacă nodul emiţător nu a primit toate mesajele de confirmare, după cateva încercari nereuşite trimite mesajul ***abort*** pentru ca mesajul trimis anterior sa fie descărcat. În acest caz, deoarece nodul care nu se mai poate contacta poate bloca funcţionarea sistemului în ansamblu, se decide eliminarea sa din sistem: toate grupurile care existau numai în acel nod sunt eliminate. Nodul respectiv nu va mai putea adresa cereri de operaţii sistemului decît după o cerere de conectare explicită. De remarcat că, prin utilizarea protocolului *two-phase commit* se asigură şi cerinţele prezentate la 3.5.2. referitoare la comunicaţiile de grup.

5.2.3.2. Varianta în inel

O altă modalitate de a realiza transmisia broadcast este prin simularea ei. Simularea se poate realiza dacă se organizează serverele locale din fiecare nod, care recepţionează cererile din reţea într-un inel logic în care se deplasează un token circulant. Evident că această organizare nu impune nici o restricţie asupra topologiei reţelei locale utilizate, care poate fi de tip Ethernet.

Numai procesul care deţine token-ul, la un moment dat, poate transmite un mesaj broadcast.

Soluţia token-ului circulant asigură ordonarea mesajelor şi un control al fluxului de mesaje în sistem. Arhitectura generală a sistemului, inclusiv structura token-ului circulant este prezentată în fig. 5.2.3.2. Cîmpul *time_stamp* identifică cel mai mic numar de secvenţă neaasociat unui mesaj din token; cînd un posesor transmite broadcast un mesaj, asociază mesajului acest numar de secvenţă şi apoi incrementează valoarea acestui cîmp. Descrierea serverului de comunicaţie este prezentată la 5.2.4.2. iar a managerului de comunicaţii în anexa A.

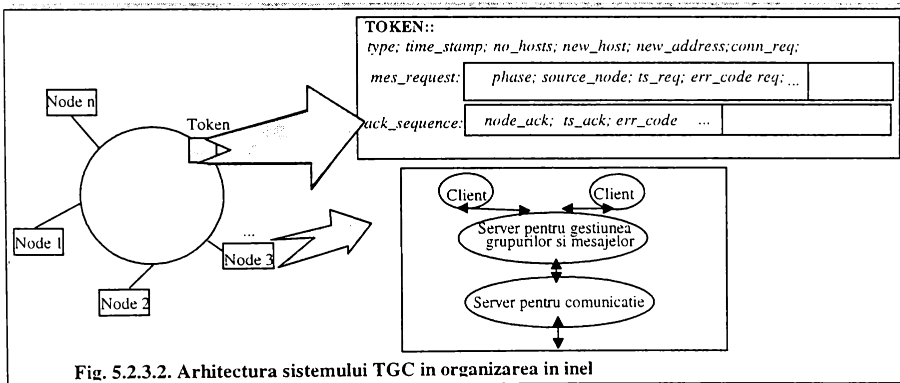


Fig. 5.2.3.2. Arhitectura sistemului TGC in organizarea in inel

5.2.4. Implementarea variantelor complet distribuite

Se va prezenta in acest subcapitol implementarea a 2 variante din cele 3 variante complet distribuite prezentate mai sus: varianta cu transmisie broadcast şi varianta cu transmisie in inel.

5.2.4.1. Implementarea variantei cu transmisie broadacast

Se va prezenta implementarea serverului de comunicaţie din arhitectura prezentată in fig. 5.2.3.1. Managerul pentru gestiunea grupurilor şi mesajelor (pe scurt GM-Group Manager) este acelaşi pentru ambele variante distribuite (varianta cu transmisie broadcast şi varianta cu transmisie în inel) şi este prezentat în anexa

BUPT

A. Pentru comunicațiile din sistem, server de comunicație <-> retea, server de comunicație <-> manager de grupuri și manager de grupuri <-> clienți se folosesc canalele reprezentate în figura 5.2.4.1.

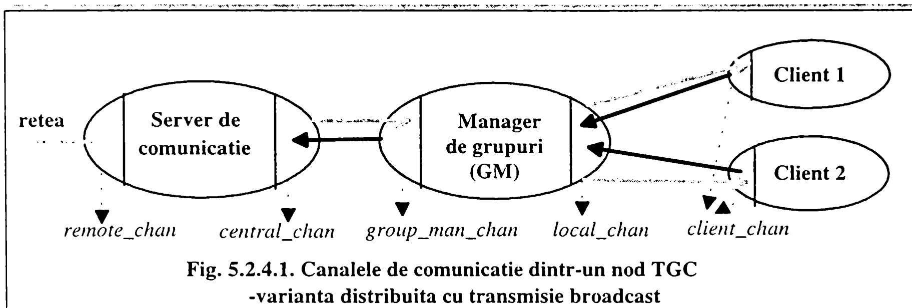


Fig. 5.2.4.1. Canalele de comunicatie dintr-un nod TGC -varianta distribuita cu transmisie broadcast

Serverul de comunicație poate fi descris astfel:

Serverul poate detecta faptul ca un nod nu mai raspunde la mesaje.
Daca detecteaza un nod care nu mai raspunde la mesaje, serverul de comunicatie il deconecteaza din sistem
Structuri de date folosite de server in plus fata de cele descrise la 5.2.2.

```
const TIME_OUT_CONNECT = ...          # timpul maxim de asteptare la conectarea la sistem
const MAX_MESSAGES = ...              # numarul maxim de mesaje ce pot fi primite de toti clientii
const WAIT_FOR_CONN=...               # timpul dupa care se repeta o cerere de conectare
const MAX_DUP=...                     # numarul maxin de retransmisii ale unui mesaj
const TIME_OUT_REQUEST=...            # timpul dupa care se repeta transmisia unei cereri

type kind = enum(CONN_REQ,, CONN_REQ_ACK)
type error = enum ( ETOO_MANY_HOSTS , EBUSY, ELOST_A_HOST, ENO_BUFS, EMORE_CLIENTS)
type mes_state= enum (free, in_transit, busy)
type mes_to_group_manager = enum ( ACK, REQ)
type MES_TO_CLIENT = record of                # buffer de mesaje pentru un client
        ( pid: int,                           # identificatorul clientului
        mes_no[1:MAX_MES_FOR_A_CLIENT]: int # indica numarul intrarii in buferul de mesaje
        pmes[MAX_MES_FOR_A_CLIENT]: pointer to MES_TO_GROUPS # indica mesajul
        pmes_state[MAX_MES_FOR_A_CLIENT]: mes_state)     # indica daca slotul este ocupat
type MESSAGE = record of                # tipul unui mesaj primit in local_chan, central_chan
        (                               #  si transamis in retea
        knd: kind,                      # tipul cererii
        client_id: int                  # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int              # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int            # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                       # lungimea partii variabile a mesajului
        mes[1:len]: char)               # mesajul
type HOSTS:   record of                 # tabela de host-uri
        (host_name[1:MAX_HOST_NAME]: char
        address: int;
        id: int;
        )
type  MES_BUF  = record of              # structura unei intrari in bufferul de mesaje
        (state: mes_state               # starea unei intrari: libera, ocupata, in tranzit
        message: MES_TO_GROUP) # cuprinde mesajul propriu zis de transmis
chan conn_chan = record of              # canalul pe care se transmite o cerere de conectare
        (knd: kind                      # tipul cererii CONN_REQ
        host_name[1:MAX_HOST_NAME]              # numele host-ului
        address:int)                            # adresa hostului
chan inout_chan [1:MAX_HOSTS] =         # canalul pe care se transmite raspunsul la o cerere de conectare
        (knd: kind,                     # cererea de conectare
```


BUPT

```
                err_code:int                # codul de eroare
                no_hosts: int               # numarul total de noduri
                hosts_tab:HOSTS,            # tabela de noduri
                no_groups: int,             # numarul de grupuri
                group_aliases_tab[1:MAX_GROUPS]:GROUP_ALIASES )# tabela de grupuri
chan group_man_chan[1:MAX_HOSTS] =  # canal pe care se transmit cererile din retea sau confirmarile
               (type: int ,                 # tipul mesajului, cerere proprie,ACK sau din retea REQ
                err_code: int,              # codul de eroare obtinut in nod in urma tratarii cererii
                mesg: MESSAGE)
chan remote_chan[1:MAX_HOSTS]       # canal broadcast pe care se transmit cererile si confirmarile in retea
               ( type: mes_to_group_manager  # tipul mesajului, cerere proprie, ACK sau REQ, din retea
                case type                   # structura mesajului transmis in retea care contine o cerere
               REQ:
                   sender: int,             # identificatorul emitatorului
                   ts:int,                  # marca de timp a mesajului
                   dup: int,                # mesaj duplicat, numarul de retransmisie
                   mes: MESSAGE)            # mesajul propriu zis primit de la GM
               ACK:                         # structura mesajului transmis in retea care contine o confirmare
                   sender: int,             # identificatorul emitatorului
                   ts_req: int,             # marca de timp a mesajului confirmat
                   ts_ack: int,             # marca de timp a confirmarii
                   err_code:int)            # codul de eroare

# informatii partajate cu managerul de grupuri GM intr-o zona de memorie partajata
var no_groups: int                                          # numarul de grupuri
var group_aliases_tab[1:MAX_GROUPS]:GROUP_ALIASES           # mentine denumirile grupurilor
var group_of_clients_tab[1:MAX_GROUPS]:GROUP_OF_CLIENTS # mentine inf. despre componenta gr-lor
var mes_to_clients_tab: [1: MAX_CLI]:MES_TO_CLIENT  # mentine inf. despre mes. in asteptare pt.clienti
var messages_buf[1: MAX_MESSAGES]: MES_BUF          # bufferul de mesaje

CommunicationServer()::
# informatii locale
var hosts: int                               # numarul de noduri conectate la sistem si de grupuri
var hosts_tab[1:MAX_HOSTS]:HOSTS:            # tabela cu informatii despre noduri
var mes_queue: queue of (knd, int, int, int, MESSAGE) # coada de cereri: tipul,sender,ts,dup,mes
var ack_list: queue of (knd, int, int, int, int)     # coada de confirmari: tipul cererii, sender, ts, err_code
var next_group_id, next_id: int              # primul identificator liber: de grup, respectiv de nod
var my_host_name[MAX_HOST_NAME], host_name[MAX_HOST_NAME]:char   # retine numele unui nod
var my_adresse, address: int          # adresa proprie si adresa altui nod primita din retea
var lc:int:=0, dup: int :=0           # marca de timp locala si nr de retransmitere al unui msaj
var err_code, err_to: int             # coduri de eroare
var mes, old_mes: MESSAGE             # cerere primita de la managerul de grupuri sau din retea
var knd: kind, ts, ts_req, ts_ack, sender, my_id     # folosite la receptia unui mesaj din retea
var ready: bool := true                      # conditioneaza emisia unui mesaj in retea
var excl: semaphore :=1        # pentru excludere reciproca in accesul la informatiile din mem. partajata
var start_GM: semaphore := 0   # pentru sincronizare server de comunicatii - manager de grupuri

initializari: my_host_name, my_address
broadcast conn_chan( CONN_REQ, my_host_name, my_address); # pt. conectare se transmite adresa
instaleaza handler-ul ALARM_function_connect pentru semnalul ALARM;
programeaza un semnal de alarma dupa un interval de TIME_OUT_CONNECT
err_code:=receive inout_chan[my_addressse](knd,err_code, hosts, host_tab, no_groups, group_aliases_tab);
if err_code = TIME_OUT_CONNECT→
     hosts = 1; initializeaza tabela hosts_tab pentru hosts = 1          # doar un singur nod in sistem
fi
fa j=1 to hosts -1 →
     receive inout_chan[my_addressse](knd,err_code, hosts, hosts_tab, no_groups, group_aliases_tab);
     insereaza, (daca nu exista) host_name, in tabela hosts_tab
af
```


BUPT

```
if err_code = TOO_MANY_HOSTS → exit                       # depasirea tabelei de host-uri
fi
instaleaza handler-ul ALARM_function_request pentru semnalul ALARM;
initializeaza hosts_tab, group_aliases_tab, group_of_clients_tab, mes_to_clients_tab
stabileste my_id ca numarul intrarii corespunzatoare nodului propriu in hosts_tab
stabileste next_group_id, next_id, ca urmatoarele intrari libere in hosts_tab, group_aliases_tab
Up(start_GM)              # managerul de grupuri poate sa transmita cereri

do true →                                          # bucla principala a serverului de comunicatii
   if not empty(conn_chan[my_address]) →
      receive conn_chan[my_address](knd, host_name, address)
      if knd = CONN_REQ →                          # receptioneaza o cerere de conectare la sistem
         if hosts < MAX_HOSTS →
           insereaza, (daca nu exista) host_name, adress in tabela hosts_tab
           actualizeaza hosts, next_id
          Down(excl)
         send inout_chan[address](CONN_REQ_ACK,SUCCESS,hosts,hosts_tab,no_groups,group_aliases_tab)
           Up(excl)
          [] hosts = MAX_HOSTS →
            Down(excl)
            send inout_chan[address](CONN_REQ_ACK,TOO_MANY_HOSTS,hosts,hosts_tab,
                                                                  no_groups, group_aliases_tab)
            Up(excl)
          fi
      fi
  fi
  if not empty(central_chan[my_address]) →         # s-a primit o cerere de la managerul de grupuri
     receive central_chan[my_address](old_mes)
     if not ready → send group_man_chan[my_address](ACK, EBUSY, old_mes) fi
     dup := 0; ready := false;
     broadcast remote_chan(REQ, my_id, lc, dup, old_mes)
     lc:= lc+1;
     programeaza un semnal de alarma care lanseaza handlerul ALARM_function_request dupa un interval
                  de TIME_OUT_REQUEST          # pentru a detecta daca s-a confirmat complet
  fi                                           # un mesaj sau trebuie retransmis
  if not empty(remote_chan[my_address]) →      # s-a primit un mesaj de la din retea
     receive remote_chan[my_address](knd)
    lc:= max(lc, ts+1); lc:= lc+1;
    if knd = REQ →                             # mesajul este o cerere
     receive remote_chan[my_address](sender, ts, dup, mes)
       if dup < MAX_DUP →                      # nu s-a depasit numarul maxim de retransmisii
         insereaza in mes_queue (mes.knd, sender, ts, dup, mes)
         err_code:= SUCCESS
         if mes.knd = LEAVE_GROUP →            # daca mai exista clienti in grup nu se sterge grupul
            Down(excl)
            if exista cel putin un client local din grupul mes.dest_group_id in group_of_clients_tab →
                 err_code:= EMORE_CLIENTS
            fi
            Up(excl)
         [] mes.knd = SEND_TO_GROUP →                  # daca nu se poate bufera mesajul, eroare
           Down(excl)
           if exista clienti locali pentru mes.dest_group_id in group_of_clients_tab →
              aloca un slot k pentru mesaj in messages_buf[k] pe care il pune in starea in_transit
              if nu se poate aloca un slot k pentru mesaj → err_code := ENO_BUFS
              [] se poate aloca un slot k pentru mesaj →
                    if nu se poate aloca un pointer in starea in_transit la message_buf[k] la un client din
                                              mes.dest_group_id parcurgind tabela mes_to_clients_tab→
```


BUPT

```
                    err_code:=ENO_BUFS
                fi
            fi
        fi
        Up(excl)
      fi
      broadcast remote_chan(ACK, my_id, ts, lc, err_code) ; lc:=lc+1   # se transmite confirmarea la cerere
    [] dup = MAX_DUP
        ready := true;
        gaseste nodul care nu a confirmat cererea cu marca de timp ts;
        elimina acel nod din hosts_tab
        fa mesajele q din mes_queue neduplicate care nu au primit confirmarea de la acel nod→
            if q.sender=my_id → send group_man_chan[my_address](ACK, ELOST_A_HOST, q.mes)
              elimina mesajul q din mes_queue
            fi
        af
    fi
  [] knd = ACK  →                                    # s-a primit o confirmare
   receive remote_chan[my_address](sender, ts_req, ts_ack, err_code)
  insereaza in ack_list: sender, ts_req, ts_ack, err_code
  if cererea cu ts_req (coresp. confirmarii) este complet confirmata (s-au primit toate confirmarile)→
       elimina din coada mes_queue cererile incomplet confirmate cu marca de timp mai mica decit ts_req
       extrage din mes_queue in q mesajul complet confirmat
       if q.sender =  my_id →
            ready := true;
           sterge programarea semnalului ALARM
       fi
       err_code := SUCCESS
       if q.knd = LEAVE_GROUP or q.knd = SEND_TO_GROUP  →
         if exista o confirmare ack la ts_req cu ack.err_code ≠ SUCCESS → err_code:=ack.err_code  fi
       fi
       if q.sender =  my_id → send group_man_chan[my_address](ACK, err_code, q.mes) fi
       [] q.sender ≠  my_id →
          if err_code= SUCCESS  →
             if q.knd = SEND_TO_GROUP
               Down(excl)
               if exista clienti locali pentru grupul q.mes.dest_group_id  →
                  send group_man_chan[my_address](REQ, err_code, q.mes)
                 dealoca mesajul din buffer_mes, pointerul din mes_to_clients_tab iar starea lor devine free
               fi
               Up(excl)
             [] err_code ≠ SEND_TO_GROUP  →
                   send group_man_chan[my_address](REQ, err_code, q.mes)
             fi
          fi
       fi
     fi
   fi
  fi
od

ALARM_function_connect()::                  # functia apelata pentru un time_out la conectare
   err_code := TIME_OUT_CONNECT
   return
ALARM_function_request()::           # functia apelata pentru un time_out la receptia tuturor confirmarilor
   if not ready →
        if dup  ≤  DUP_MAX →
             dup := dup+1;
```


BUPT

```
            send remote_chan[my_address](my_id, lc, dup, old_mes);  # se transmite o cerere duplicat
            lc := lc + 1;
            programeaza un semnal de alarma care lanseaza handlerul ALARM_function_request
            dupa un interval de timp de TIME_OUT_REQUEST
        fi
fi
return
```

5.2.4.2. Implementarea variantei cu transmisie în inel

Se va prezenta implementarea serverului de comunicaţie din arhitectura prezentată in fig. 5.2.3.2. Managerul pentru gestiunea grupurilor şi mesajelor (pe scurt GM-Group Manager) este acelaşi pentru ambele variante distribuite (varianta cu transmisie broadcast şi varianta cu transmisie în inel) şi este prezentat în anexa A. Pentru comunicaţiile din sistem, server de comunicaţie <-> retea, server de comunicaţie <-> manager de grupuri şi manager de grupuri <-> clienţi se folosesc canalele reprezentate în figura 5.2.4.2.

Serverul de comunicaţie poate fi descris astfel:

```
# Serverul de comunicatie este specific variantei cu transmisie in inel
# Serverul poate detecta faptul ca un nod nu mai raspunde la mesaje.
```

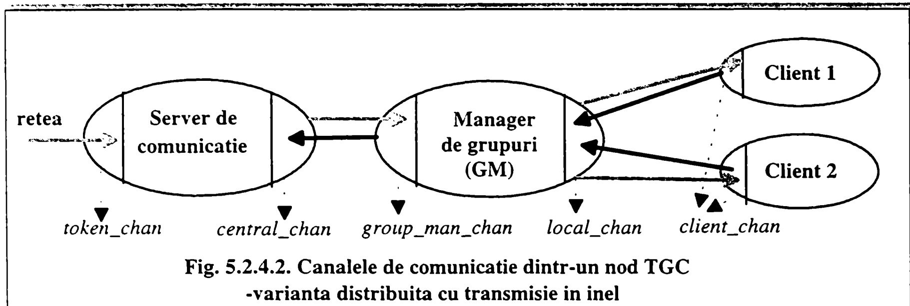


Fig. 5.2.4.2. Canalele de comunicatie dintr-un nod TGC -varianta distribuita cu transmisie in inel

```
# Daca detecteaza un nod care nu mai raspunde la mesaje, serverul de comunicatie il deconecteaza din sistem

const MAX_MESSAGES = ...           # numarul maxim de mesaje ce poate fi primit de toti clientii
const WAIT_FOR_CONN=...            # timpul dupa care se repeta o cerere de conectare
const TIME_OUT_CONNECT = ...       # timpul maxim de asteptare la conectarea la sistem
const TIME_OUT_TOKEN=...           # timpul maxim cit se asteapta token-ul
const TIME_OUT_TEST =...           # timpul maxim cit se asteapta un mesaj de test

type kind = enum(NAME_REQ., NAME_REQ_ACK, CONN_REQ, CONN_REQ_ACK, NEW_TOKEN, TEST)
type mes_to_group_manager = enum ( ACK, REQ)
type mes_state=(free, in_transit, busy)
type error = enum ( ETOO_MANY_HOST , ETOO_MANY_REQ,ETOO_MANY_REQ_IN_NETW, EBUSY)
type MES_TO_CLIENT = record of                   # buffer de mesaje pentru un client
        ( pid: int,                              # identificatorul clientului
        mes_no[1:MAX_MES_FOR_A_CLIENT]: int # indica numarul intrarii in buferul de mesaje
        pmes[MAX_MES_FOR_A_CLIENT]: pointer to MES_TO_GROUPS # indica mesajul
        pmes_state[MAX_MES_FOR_A_CLIENT]: mes_state)        # indica daca slotul este ocupat
type MESSAGE = record of               # tipul unui mesaj primit in local_chan, central_chan
        (                              #  si transamis in retea
        knd: kind,                     # tipul cererii
        client_id: int                 # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int             # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int           # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                      # lungimea partii variabile a mesajului
        mes[1:len]: char)# mesajul
```


BUPT

```
type ACK_TYPE = record of                # structura unei confirmari la o cerere in token
            ( node_ack: int              # adresa proprie a nodului care confirma o cerere
              ts_ack: int                # marca de timp a cererii confirmate
              err_code: int)             # codul de eroare obtinut la confirmarea unei cereri
type MES_BUF = record of                 # structura unei intrari in bufferul de mesaje
            (state: mes_state            # starea unei intrari: libera, ocupata, in tranzit
             message: MES_TO_GROUP)      # mesajul propriu zis transmis
type TOKEN = record of
            (type: int                               # tipul mesajului TOKEN sau TEST
             time_stamp: int                         # marca de timp a mesajelor
             no_hosts: int                           # numarul de noduri
             max_req: int                            # numarul de cereri din token
             max_ack: int                            # numarul de confirmari din token
             new_host[1:MAX_HOST_NAME]: char         # numele nodului nou conectat
             new_address: int                        # adresa nodului nou conectat
             conn_req: int                           # folosit la conectarea unui nod
             mes_request[1:MAX_HOSTS*2]: record of
                   phase: int;
                   source_node: int
                   ts_req: int,
                   err_code: int,
                   req: MESSAGE)                     # coada de cereri
             ack_sequence: [1:MAX_HOSTS*MAX_HOSTS*2]: ACK_TYPE   # coada de confirmari
            )
chan token_chan[1..MAX_HOSTS](token: TOKEN) # canalul pe care se transmit mesajele in inelul logic
chan inout_chan [1..MAX_HOSTS] =      # canalul pe care se transmite o cerere de conectare si validarea la
            (knd: kind                # cererea de conectare
             case knd
              NAME_REQ_ACK, CONN_REQ,:
                  no_hosts: int                          # numarul nodurilor
                  host_name[1:MAX_HOST_NAME]             # numele host-ului
                  address:int                            # adresa hostului
             CONN_REQ_ACK:
                  err_code: int                          # codul de eroare
                  token:TOKEN                            # token-ul primit la conectare
                  no_groups: int,
                  group_aliases_tab[1:MAX_GROUPS]:GROUP_ALIASES )
            )
chan name_chan [1..MAX_HOSTS] =       # canalul pe care se transmite o cerere de deconectare a unui nod
            (                         # din sistem sau un mesaj care anunta un nod care nu raspunde
             knd: kind                               # tipul cererii
             host_name[1:MAX_HOST_NAME]              # numele host-ului
             address:int)                            # adresa hostului
chan group_man_chan: [1..MAX_HOSTS] = # canal pe care se transmit cereri din retea sau confirmari la GM
            (type: mes_to_group_manager  # tipul mesajului, cerere proprie, ACK sau REQ, din retea
             err_code: int               # codul de eroare
             mesg: MESSAGE)
chan central_chan [1:MAX_HOSTS] =     # canal pe care se primesc cererile de la managerul de grupuri(GM)
            (mes:MESSAGE)

# informatii partajate cu managerul de grupuri intr-o zona de memorie partajata
var no_groups: int                                        # numarul de grupuri
var group_aliases_tab[1:MAX_GROUPS]:GROUP_ALIASES         # mentine denumirile grupurilor
var group_of_clients_tab[1:MAX_GROUPS]:GROUP_OF_CLIENTS # mentine inf. despre componenta gr-lor
var mes_to_clients_tab: [1: MAX_CLI]:MES_TO_CLIENT    # mentine inf. despre mes. in asteptare pt.clienti
var messages_buf[1: MAX_MESSAGES]: MES_BUF   # bufferul de mesaje
```


BUPT

```
CommunicationServer()::
# informatii locale
var hosts: int                                # numarul de noduri conectate la sistem
var pending_req[1:MAX_REQUESTS]: MESSAGE      # bufferul de cereri memorate pina la primirea token-ului
var hosts_tab[1:MAX_HOSTS]:record of (        # tabela de host-uri
        host_name[1:MAX_HOST_NAME]: char
        address: int)
var token:TOKEN                           # variabila care pastreaza token-ul
var my_host_name[MAX_HOST_NAME], host_name[MAX_HOST_NAME]:char # retine numele unui nod
var my_adresse, next_address, pred_addresss, address: int        # adresa proprie si a nodului vecin
var new_conn: bool = false                # indica faptul ca s-a primit o cerere de conectare
var knd: kind, err_code: int              # tipul unei cereri si  codul de eroare
var mes: MESSAGE                          # cerere primita de la GM
var excl: semaphore :=1          # pentru excludere  reciproca in accesul  la informatiile din mem. partajata
var start_GM: semaphore := 0     # pentru sincronizare server de comunicatii - manager de grupuri

initializari: my_host_name, my_address
again:broadcast  name_chan( NAME_REQ, my_host_name, my_address);      # cerere de conectare la sistem
instaleaza  handler-ul ALARM_function_connect  pentru semnalul ALARM;
programeaza un semnal de alarma dupa un interval de TIME_OUT_CONNECT
err_code := receive inout_chan[my_addressse](knd, hosts, host_name, address);
if err_code = TIME_OUT_CONNECT →                 # numai un singur nod in sistem
      hosts = 1;
      initializeaza  tabela hosts_tab pentru  hosts = 1
fi
fa j=1 to hosts -1 →
     receive inout_chan[my_address](knd)
      if knd = NAME_REQ_ACK → receive inout_chan[my_address]( hosts, host_name, address); fi
      insereaza, (daca nu exista) host_name, address in tabela hosts_tab (tabela sa fie ordonata dupa adrese IP)
fa
alege numele predecesorului in inel;
stabileste nodul succ in inel  next_address si cel pred pred_address astfel incit tabela sa fie ordonata
send inout_chan[pred_addressse](CON_REQ,,0, my_host_name, my_address);
receive inout_chan[my_address] (knd, err_code);
if err_code = EBUSY→
   sleep (WAIT_FOR_CONN); goto again
[] err_code = ETOO_MANY_HOST →
    exit;
[] receive inout_chan[my_addressse] (token, no_groups,group_aliases_tab);
fi
instaleaza  handler-ul ALARM_function_token  pentru semnalul ALARM;
initializeaza group_of_clients_tab, mes_to_clients_tab
initializeaza bufferul de mesaje messages_buf,  bufferul de cereri pending_req
token.time_stamp := token.time_stamp+1
insereaza pentru toate cererile din mes_request din token o confirmare cu
          source_ node:=my_address, ts_ack:=mes_request[j].ts, err_code:=SUCCESS
send  token_chan [next_address](token)
programeaza un semnal de alarma (pentru cazul cind nu se receptioneaza token-ul)
                                                        dupa un interval de TIME_OUT_TOKEN
Up(start_GM)              # managerul de grupuri poate sa transmita cereri

do true →                        # bucla principala a serverului de comunicatii
  if not empty(name_chan[my_address]) →
     receive name_chan[my_address](knd,host_name, address)
     if  knd = NAME_REQ →       # receptioneaza o cerere de livrare a numelui si a adresei nodului
          send inout_chan[address](NAME_REQ_ACK,hosts, my_host_name, my_address)
     [] knd = NEW_TOKEN  →  # genereaza un nou token la pierderea token-ului
          actualizeaza  hosts_tab, hosts:=hosts-1
          if my_host_name precede in inel pe host_name →
```


BUPT

```
            token.hosts:= token.hosts - 1
            elimina din token.ack_sequence toate confirmarile date de host_name
            actualizeaza next_address la nodul urmator nodului host_name in hosts_tab
              send token_chan[next_address](token);
            programeaza un semnal de alarma (pentru cazul cind nu se receptioneaza token-ul)
                                                          dupa un interval de TIME_OUT_TOKEN
        fi
    fi
fi
if not empty(inout_chan[my_address]) →
    receive inout_chan[my_address](knd)  # receptioneaza o cerere de conectare in sistem
    if knd =CONN_REQ →
      receive inout_chan[my_address](temp,host_name, address); new_conn = true;
    fi
    if hosts = 1 →
        actualizeaza next_address, deci nodul succesor in inel ; hosts = hosts +1
        insereaza, (daca nu exista) host_name, adress in tabela hosts_tab astfel incit tabela sa fie ordonata
                                                                   in functie de adresele IP
        token.new_host:=NULL; token.new_address:=NULL;
        token.no_hosts:=  hosts; token.time_stamp:= 1; token.conn_req:=0
        Down(excl)
        send inout_chan[next_address](CONN_REQ_ACK,SUCCESS,token,no_groups,group_aliases_tab)
        Up(excl)
    fi
fi
if not empty (token_chan[my_address])    # receptioneaza token-ul
    receive token_chan[my_address] (token);  hosts:= token.no_hosts;
   fa mesajele j din token.mes_request in ordinea marcilor de timp →
     if  token.mes_request[j].phase = 2 →
         err_code :=token.mes_request[j].err_code
         if  token.mes_request[j].source_node = yAddresse  →        # cerere locala
              send group_man_chan[my_address](ACK, err_code, token.mes_request[j].req)
              Down(excl)
              if err_code=SUCCESS and  token.mes_request[j].req.knd= SEND_TO_GROUP
                                    and exista cel putin un client  in group_of_clients_tab din grupul
                                                   token.mes_request[j].req.dest_group_id→
               dealoca intrarile marcate in_transit din mes_to_clients_tab  si  messages_buf
             fi
             Up(excl)
         [] token.mes_request[j].source_node ≠ myAddresse  →       # cerere din retea
           if err_code=SUCCESS →
             if  token.mes_request[j].req.knd= SEND_TO_GROUP →
              Down(excl)
              if  exista cel putin un client  in group_of_clients_tab din grupul
                                                   token.mes_request[j].req.dest_group_id
                       send group_man_chan[my_address](REQ, err_code, token.mes_request[j].req)
                       dealoca intrarile marcate in_transit din mes_to_clients_tab  si  messages_buf
              fi
              Up(excl)
             [] token.mes_request[j].req.knd ≠  SEND_TO_GROUP →
              send group_man_chan[my_address](REQ, err_code, token.mes_request[j].req)
             fi
           fi
       fi
     [] token.mes_request[j].phase = 1 →
         aloca un slot l in token pentru confirmarea  cererii token.mes_request[j]
          token.ack_sequence[j][l].node_ack := my_address;
          token.ack_sequence[j][l].ts_ack := token.mes_request[j].ts
```


BUPT

```
            if token.mes_request[j].req.knd= SEND_TO_GROUP →
              Down(excl)
              if exista clienti locali pentru token.mes_request[j].req.dest_group_id→
                aloca un slot k  pentru mesaj in message_buf [k] pe care il pune in starea in_transit
                if nu se poate aloca un slot k pentru mesaj  →
                  token.ack_sequence[j][l].err_code := ENO_BUFS
                  token.mes_request[j].err_code:=ENO_BUFS
                [] se poate aloca un slot k pentru mesaj →
                  if nu se poate aloca un pointer in starea in_transit  la message_buf[k] la un client din
                           token.mes_request[j].dest_group_id parcurgind tabela mes_to_clients_tab→
                      token.ack_sequence[j][l]err_code := ENO_BUFS
                      token.mes_request[j].err_code:=ENO_BUFS
                  fi
                fi
              fi
              Up(excl)
            [] token.mes_request[j].req.knd= LEAVE_GROUP →
              Down(excl)
              if  exista cel putin un client  in group_of_clients_tab din grupul
                                              token.mes_request[j].req.dest_group_id→
                  token.ack_sequence[j][l]err_code := EMORE_CLIENTS
                  token.mes_request[j].err_code:= EMORE_CLIENTS
              fi
              Up(excl)
            fi
    fi
    if token.mes_request[j].source_node = myAddresse  →          # cerere lansata de nodul respectiv
        token.mes_request[j].phase:= token.mes_request[j].phase+1
        if  token.mes_request[j].phase = 3 →
            dealoca mes_request[j] din token si ordoneaza mesajele in coada
            dealoca confirmarile pentru  mes_request[j] din coada de confirmari din token
        fi
    fi
  af
  if exista loc k in token.mes_request pentru urmatoarea cerere  j din pending_req[j]→
    copiaza cererea  pending_req[j] in  token.mes_request; dealoca intrarea din  pending_req[j]
    token.mes_request[k].err_code := SUCCES; token.mes_request[k].phase:= 1;
    token.mes_request[k].ts:=token.time_stamp
  [] nu exista loc in token.mes_request pentru o cerere →
    send group_man_chan[my_address](ACK,ETOO_MANY_REQ_IN_NETW, pending_req[j])
    dealoca intrarea din  pending_req[j]
  fi
  if (new_conn = true) →
    if  (token.conn_req = 0)→
      if  hosts = MAX_HOSTS →
        send  inout_chan[address](CON_REQ_ACK, TOO_MANY_HOSTS);
        send token[next_address](token);
      [] hosts < MAX_HOSTS→
        actualizeaza next_address, deci nodul succesor in inel
        insereaza, (daca nu exista) host_name, adress in tabela hosts_tab (tabela ordonata in fct de adrese IP)
        token.new_host:=host_name; token.new_address:=address;
        token.no_hosts:=  token.no_hosts + 1;
        token.time_stamp:=token.time_stamp+1;
        token.conn_req:=token.conn_req+1
        send inout_chan[next_address](CONN_REQ_ACK,SUCCESS,token,no_groups,group_aliases_tab)
      fi
    [] token.conn_req <  token.no_hosts→
      send  inout_chan[address](CONN_REQ_ACK, EBUSY)
```


BUPT

```
            insereaza, (daca nu exista) token.new_host,token.new_adress in tabela hosts_tab
            hosts := token.no_hosts; new_conn:= false; token.conn_req:=token.conn_req+1
            token.time_stamp:=token.time_stamp+1;
             send token[next_address](token)
        [] token.conn_req = token.no_hosts→
             new_conn:=false; token.conn_req := 0
             token.time_stamp:=token.time_stamp+1;
              send token[next_address](token)
        fi
    [] (new_conn = false) →
        token.time_stamp:=token.time_stamp+1;
        if token.conn_req > 0→
          insereaza, (daca nu exista) host_name, adress in tabela hosts_tab;
           token.conn_req:=token.conn_req+1
        fi
         send token[next_address](token);
    fi
  fi
  if not empty (central_chan[my_address])→                  # cerere de la managerul de grupuri
    receive central_chan[my_address](mes);
    if hosts = 1 → send group_man_chan[my_address](ACK, SUCCESS, mes)
    [] if exista loc in pending_request →    # cererea se bufereaza pentru a fi transmisa la primirea token-ului
          aloca urmatorul slot si memoreaza mes
       [] nu exista loc in pending_request →
          send  group_man_chan[my_address](ACK, ETOO_MANY_REQ, mes)
                                                               # daca nu exista loc cerere este refuzata
       fi
    fi
  fi
od

ALARM_function_connect()::                 # functia apelata pentru un time_out la conectare
   err_code := TIME_OUT_CONNECT
   return

ALARM_function_token()::                   # functia apelata pentru un time_out la primirea token-ului
     token.type := TEST
     send token[next_address](token)
      programeaza un semnal de alarma care lanseaza handlerul ALARM_function_disconnect
              dupa un interval de timp de TIME_OUT_REQUEST
      receive token[my_address](token)
     return

ALARM_function_disconnect()::              # functia apelata pentru un time_out la primirea token-ului de test
     broadcast  name_chan( NEW_TOKEN, nume nod care precede pe my_host_name,
                                                 adresa nod care precede pe my_address)
     return
```

5.2.5. Comparație între variantele propuse și concluzii

În urma analizei celor patru variante propuse, varianta 1- varianta semidistribuită, varianta a 2 a - varianta distribuită cu mesaje broadcast, varianta a 3 a - varianta distribuită fără utilizare broadcast, varianta a 4 a - varianta în inel, se pot enunța următoarele concluzii:

(1) În toate soluțiile se consideră că se cunoaște numărul participanților în sistem pentru a se putea lua decizia dacă un mesaj a fost recepționat sau nu de *toți* membrii unui grup. În variantele 1 și 3 trebuiecunoscută și identitatea participanților (nodurilor). În varianta a 4 a trebuie cunoscută identitatea vecinilor din inel.

(2) Conectarea unui nod nou la sistem se poate face numai după confirmarea cererii de conectarea a acestuia de către toate nodurile în variantele 2, 3, după confirmarea serverului central în varianta 1 și după confirmarea de către nodurile vecine în inel în varianta a 4 a. În variantele 2, 3 și 4, la conectare, nodul

BUPT

conectat trebuie să-şi construiască şi tabela care menţine legătura *group_id* - *alias*-uri, eventual şi dispunerea grupurilor pe noduri (varianta a 3 a).

(3) Implementarea Unix şi Windows a descrierii variantei 1 prezentate la 5.2.2.3. este directă: pentru serverele locale se foloseşte o implementare cu *select*, pentru serverul central master se foloseşte în Unix *fork+exec* şi o zonă de memorie partajată între serverul central master şi toate serverele centrale fii iar în Windows serverele centrale se implementează cu fire.

(4) Atomicitatea se obţine folosind pentru comunicaţia în reţea protocolul TCP şi fie folosind direct un protocol *two-phase-commit* (varianta a 3 a), fie un protocol echivalent (în varianta a 4 a se simulează *two-phase-commit* în inelul logic al procesoarelor). În varianta a 2 a se foloseşte protocolul UDP pentru a putea beneficia de transmisia broadcast iar atomicitatea se obţine datorită faptului că fiecare nod "cunoaşte" toate confirmările sosite pentru un mesaj.

(5) În cazul în care un nod este deconectat de la sistem fără a fi emis o cerere de deconectare, şi deci refuză să mai primească mesaje, în toate variantele se forţează automat, pentru nodul respectiv, o cerere de deconectare de la sistem. În varianta a 4 a trebuie reconstruit inelul logic al procesoarelor.

(6) În ceea ce priveşte o comparaţie a sistemului TGC proiectat, cu alte sisteme de comunicaţii de grup se pot face următoarele observaţii: sistemul Totem [MMA96] foloseşte soluţia organizării procesoarelor în inel, sistemele MCL şi S/Net Linda Kernel folosesc transmisie broadcast, iar sistemul Vartlaap [LRM93] o soluţie semidistribuită. Recent, unele sisteme de comunicaţii de grup au fost suplimentate cu facilităţi pentru toleranţa la defecte. Sistemul TGC, deşi suportă deconectarea din sistem a unui nod nu este prevăzut cu o completă toleranţă la defecte, în special în prima variantă în care serverul central este unic. Toleranţa la defecte în prima variantă impune menţinerea unei copii a serverului central (mecanismul de "mirroring" al serverului central).

(7) În toate variantele, identificatorii de grup, globali în întreg sistemul, sunt refolosiţi după distrugerea unui grup, la părăsirea acestuia de către ultimul membru. Se apreciază că, la o extindere ulterioară a sistemului TGC se poate lua în considerare observaţia ca identificatorii de grup acordaţi noilor grupuri să aibă valori nefolosite (asemănător *pid*-urilor din Unix).

(8) În toate variantele alegerea unui identificator pentru un grup trebuie făcută în excludere reciprocă. Prin urmare, execuţia operaţiei *Create_Group* trebuie făcută în regim de excludere reciprocă folosind unul din algoritmii prezentaţi şi implementaţi în subcapitolele 4.3.2., 4.3.3., 4.3.4. În prima variantă, această problemă se pune doar local deoarece serverele centrale fii aleg identificatorii de grup; accesul la identificatori se face controlat de un semafor local.

(9) În ceea ce priveşte primitivele sistemului TGC acestea pot fi extinse astfel: (a) primitiva *SendToGroup()* poate fi extinsă astfel încît să admită ca destinaţie şi grupul *ALL* (în sensul generic de toate grupurile). (b) poate fi introdusă o nouă primitivă care să găsească toate procesele compoente ale unui grup, la un moment dat. (c) un grup poate să rămînă în folosinţă chiar dacă nici un proces nu mai este conectat la el; aceasta implică introducerea unei noi primitive pentru distrugerea explicită a unui grup.

(10) Sucesiunea de cereri adresate de un client sistemului TGC, în oricare din cele trei variante pentru conectarea la un grup este următoarea:

```
Client[i:1..n]::
  var done: bool, code: int                            #variabile de lucru
  var MyGroupId: int = 0                               # identificator de grup
  var MyGroupAlias[1: MAX_NAME_LEN]: char = ...        # numele grupului
  var MyName[1: MAX_NAME_LEN]: char = ...              #numele procesului

      done := false;
      InitTGC(MyName, NULL, NULL)
      do not done →
              code := GetIdGroup(MyGroupAlias, MyGroupId)
              if code = SUCCESS →
                      code := JoinToGroup(MyGroupId);
                      if code = SUCCESS → done := true fi
              [] code ≠ SUCCESS →
                      code := CreateGroup(MyGroupAlias, MyGroupId)
                      if code = SUCCESS → done := true fi
              fi
      od
```


BUPT

Capitolul 6.

SISTEME DE PROGRAMARE DISTRIBUITĂ BAZATE PE OBIECTE (SPDBO). CRITERII DE PROIECTARE

6.1. Concepte fundamentale

Un *obiect* este definit ca o entitate care încapsulează informaţii de stare (sau date) şi un set de operaţii sau proceduri care manipulează aceste date într-un mod controlat, astfel încît pot fi tratate, colectiv, ca o singură unitate (fig 6.1).

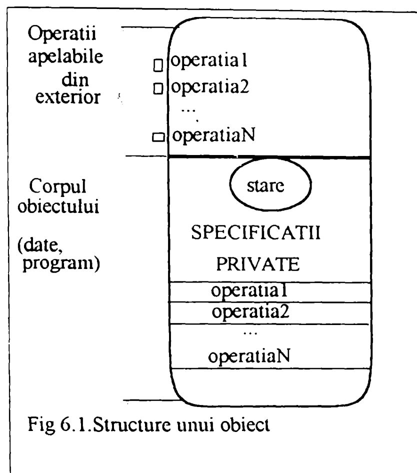


Fig 6.1.Structure unui obiect

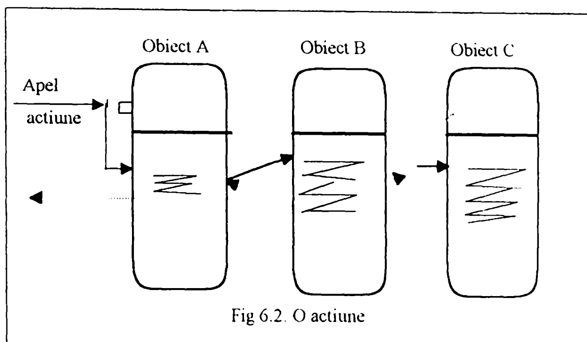


Fig 6.2. O actiune

Singura modalitate în care poate fi modificată sau examinată starea unui obiect este prin apelul operaţiilor facute publice, deci accesibile din exterior. Aceasta crează o interfaţă exterioară pentru obiect, prin care se specifică operaţiile accesibile asupra obiectului pastrînd însă implementarea lor şi informaţiile de stare inaccesibile în exterior.

O operaţie asupra unui obiect poate apela alte operaţii, posibil chiar asupra altor obiecte; aceste operaţii la rîndul lor pot apela operaţii asupra altor obiecte s.a.m.d. O inlănţuire a unor astfel de apeluri de operaţii se numeşte *acţiune* (fig 6.2.).

Limbajele de programare pot încuraja lucrul cu obiecte, permitînd dezvoltarea unor programe constind din obiecte autonome care interacţioneaza: astfel de limbaje pot fi clasificate [CC91] în *limbaje bazate pe tipul obiect*, pentru care tipul obiecteste o caracteristică a limbajului: Ada, Modula-2 şi *limbaje orientate pe obiect* (care implementează şi conceptul de moştenire): C++, SR şi Smalltalk. Conform principiilor urmărite de aceste limbaje, un program trebuie să aibă urmatoarele cinci caracteristici [CC91]:

1. ***să fie abstract.*** Concepţiile majore de proiectare trebuie să fie separate de detaliile de implementare.
2. ***să fie structurat.*** Un program complex trebuie să poată fi descompus în mai multe componente de dimensiuni adecvate şi între care să poată fi stabilite relaţii bine definite.
3. ***să fie modular.*** Structura internă a fiecărei componente trebuie să fie astfel concepută încît să nu depindă de structura altei componente.
4. ***să fie concis.*** Codul programului trebuie să fie cît mai clar.
5. ***să fie verificabil.*** Programul trebuie să fie uşor de testat şi depanat.

Un ***sistem de programare bazat pe obiecte (SPBO)*** poate fi definit ca un mediu de calcul furnizat de un limbaj de programare bazat pe obiecte şi de un sistem de operare sau o extensie a sa care implementează conceptul de obiect. Aceasta permite obiectelor să fie manipulate, gestionate şi utilizate eficient, dar şi partajarea obiectelor de către mai mulţi utilizatori; ultima caracteristică nu este asigurată de limbajele de programare. Sistemul de operare dintr-un SPBO furnizează facilităţi în ceea ce priveşte mai ales interacţiunea obiectelor şi gestiunea resurselor.

Un ***sistem de programare distribuit bazat pe obiecte (SPDBO)*** reuneşte caracteristicile unui sistem de programare bazat pe obiect ca şi cele ale unui sistem de calcul distribuit. Aceste caracteristici sunt [CC91]:

1. ***Distribuire.*** Un SPDBO uneşte nodurile de lucru independente ale unei reţele locale de calculatoare, posibil chiar eterogene astfel încît să furnizeze un sistem de calcul descentralizat.
2. ***Transparenta.*** Un SPDBO trebuie să ascundă caracteristică de distribuţie a mediului de calcul faţă de utilizatori. De exemplu, localizarea obiectelor în reţea trebuie să fie transparentă pentru utilizatori şi pe de altă parte trebuie să furnizeze acces uniform la toate obiectele sistemului, fie că aceste obiecte sunt active în memorie sau inactive în memoria secundară.

BUPT

3. ***Integritatea datelor.*** Într-un SPDBO, înaintea execuției oricărei operații asupra sa, un obiect trebuie să fie într-o stare validă. Un obiect este într-o *stare validă* dacă starea sa a rezultat în urma execuției cu succes a unei operații. Dacă o operație nu se execută cu succes, atunci sistemul trebuie să anuleze toate modificările survenite în cadrul acelei operații.

4. ***Toleranța la defecte.*** Apariția unui defect al unui nod sau al unui obiect trebuie să reprezinte pentru SPDBO numai un defect parțial; restul sistemului trebuie să fie capabil să continue prelucrările, eventual cu inconvenientul lipsei unui serviciu.

5. ***Recuperabilitate.*** În urma apariției unui defect al unui nod și înlăturării lui, un SPDBO trebuie să fie capabil să recupereze obiectele permanente de pe acel nod.

6. ***Autonomia obiectelor.*** Un SPDBO poate permite proprietarului unui obiect să specifice clienții cărora li se permite să execute operații asupra acelui obiect.

7. ***Concurența la nivelul programelor.*** Un SPDBO trebuie să furnizeze programelor posibilitatea ca operațiile asupra diferitelor obiecte să se execute concurent, în procesoare diferite. (fig 6.3.)

8. ***Concurența la nivelul obiectelor.*** Asupra unui obiect pot să se execute concurent mai multe operații, evident cu respectarea restricțiilor de consistență ale obiectului.

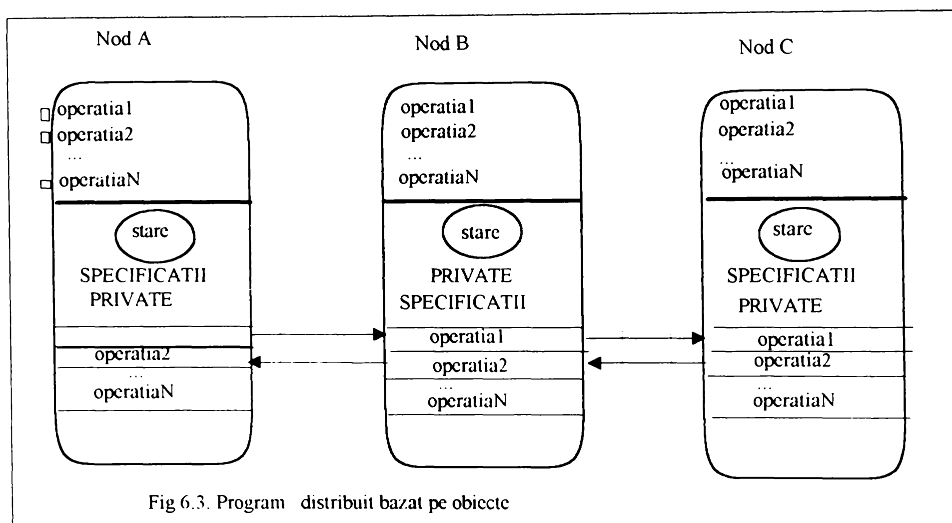


Fig 6.3. Program distribuit bazat pe obiecte

6.2. Structura obiectelor

Dimensiunea obiectelor și cantitatea de prelucrări executate asupra lor caracterizează ***granulitatea obiectelor.*** În cel mai simplu caz un SPDBO permite numai obiecte complexe, de dimensiune mare, cu operații conținînd un număr mare de instrucțiuni și cu relativ puține interacțiuni cu alte obiecte. Exemple de astfel de obiecte sunt un fișier sau o baza de date monoutilizator. Caracteristic acestor obiecte este faptul că sunt rezidente în spațiul propriu de adrese.

Pentru un control mai eficient al datelor, un SPDBO poate permite și obiecte de complexitate medie, cu exponenți tipici listele înlănțuite și componentele unei baze de date multiutilizator. Aceste obiecte pot fi rezidente în spațiul de adrese al obiectelor complexe. Aceasta permite un grad mai ridicat de concurență, deoarece sincronizarea accesului la date se face la nivelul obiectelor de complexitate medie, dar și reducerea cantității de date copiate în memoria secundară atunci cînd o acțiune este desavîrșită *(commits)*. Obiectele de complexitate mică sunt caracterizate prin dimensiune mică, operații compuse dintr-un număr mic de instrucțiuni și număr mare de interacțiuni cu alte obiecte. Exemple specifice sunt tipurile de date standard furnizate de limbajele de programare: numere întregi, valori logice, numere complexe. Un astfel de obiect este rezident în spațiul de adresă al unui obiect de compexitate medie sau mare.

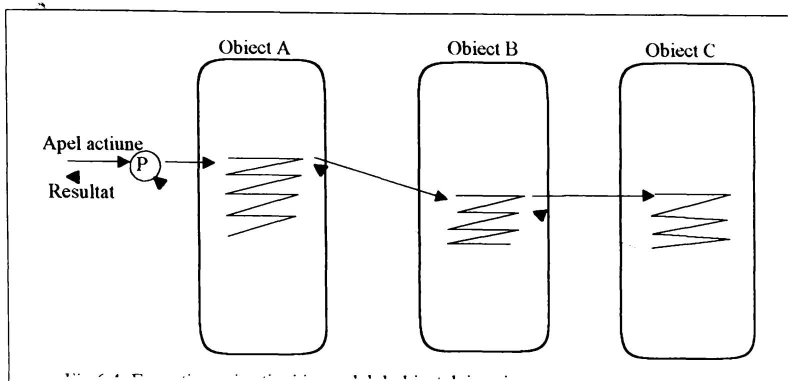
Structura obiectelor mai este decisă și de relațiile dintre procese și obiecte. Procesele pot fi implementate separat și asociate temporar obiectelor pe care le examinează sau modifică sau pot fi asociate permanent obiectelor respective. Aceste două implementări corespund celor două modele cunoscute: ***modelul obiectului pasiv*** și ***modelul obiectului activ.***

În ***modelul obiectului pasiv*** procesele și obiectele sunt entități complet separate. Un proces nu este restricționat la un singur obiect, dar numai într-un singur proces sunt inglobate toate operațiile dintr-o acțiune.

În consecință, un proces se poate executa asupra mai multor obiecte pe toată perioada existenței sale. Cînd dintr-un proces se apelează o operație asupra unui obiect, execuția procesului se comută, migrează în obiectul

BUPT

respectiv. Dupa terminarea execuţiei operaţiei, procesul este returnat la operaţia apelantă; un exemplu este prezentat în figura 6.4.:



În *modelul obiectului activ* se crează unul sau mai multe procese servere care se asignează obiectului şi care implementează operaţiile ce se pot executa asupra obiectului. Fiecare proces este deci restricţionat la un obiect particular; atunci cînd este distrus obiectul, sunt distruse şi procesele server asociate lui. Interacţiunea dintre un proces server şi un proces client se desfăşoară în modelul obiectului activ astfel [CC91] [DG89]:

(1) Procesul client prezintă cererea de execuţie a unei operaţii împreuna cu o listă de argumente;

(2) Procesul server acceptă cererea, apoi localizează şi execută operaţia specificata;

(3) Dacă în cursul execuţiei operaţiei se apelează o operaţie asupra unui alt obiect, procesul server, devenit acum proces client prezinta cererea şi asteapta rezultatul.

(4) Cînd execuţia operaţiei se termină, procesul server returnează rezultatul clientului.

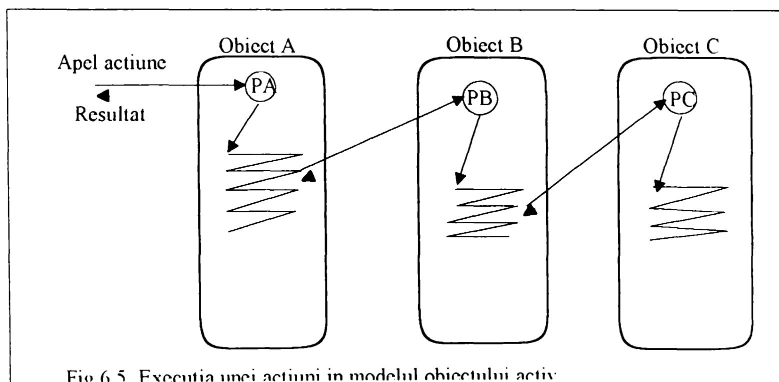


Fig.6.5. Executia unei actiuni in modelul obiectului activ

În acest model al obiectului activ, în execuţia unei singure acţiuni pot fi implicate mai multe procese (fig.6.5.):

În acest model, al obiectului activ, într-o singură acţiune pot fi implicate mai multe procese.

În *varianta statică a modelului activ*, la creare, fiecărui obiect i se asociaza un număr fix de procese server. Orice cerere de execuţie a unei operaţii asupra unui obiect este directionată către un proces server liber. Dacă toate procesele server sunt ocupate, cererile sunt inlantuite într-o coada de asteptare, urmînd a fi servite în momentul cînd serverele devin disponibile. Gradul de multiprogramare obţinut prin aceasta varianta este, evident, limitat de numărul de procese server create pentru un obiect (vezi 3.1.2.).

În *varianta dinamică a obiectului activ*, se crează cîte un proces server pentru fiecare obiect, atunci cînd se recepţionează o cerere de execuţie a unei operaţii asupra acelui obiect. La terminarea execuţiei, procesul respectiv este distrus. Astfel, în aceasta varianta nu se inlantuie cererile de execuţie a operaţiilor asupra obiectelor, însă cu dezavantajul unui timp aditional necesar pentru crearea şi distrugera dinamică a proceselor (vezi 3.1.2.).

Modelul obiectului activ introduce posibilitatea apariţiei interblocarilor. O interblocare poate apare ori de cîte ori nu există procese server suficiente pentru a executa cererile de operaţii recepţionate pentru obiectul respectiv. De exemplu, în varianta statică o singură acţiune poate apela acelaşi obiect de mai multe ori decît numărul de procese server asociate obiectului (o astfel de situaţie poate fi un apel recursiv); în momentul în care nu mai există procese server disponibile apare interblocarea. În varianta dinamica, interblocarea este mai puţin o problemă; ea poate apare totuşi dacă se permite ca o acţiune să aibă un număr oarecare de apeluri de operaţii.

BUPT

6.3. Gestionarea obiectelor

Acest subcapitol descrie pe scurt caracteristicile unui SPDBO în ceea ce privește permanentizarea efectelor unei acțiuni asupra obiectelor, sincronizarea execuțiilor concurente de apeluri asupra unui obiect, protejarea obiectelor împotriva clienților neautorizati și recuperarea obiectelor în cazuri de defect.

6.3.1. Gestionarea acțiunilor

Acțiunile trebuie să aibă urmatoarele trei proprietăți ([CC91]):

1. ***Seriabilitate*** Mai multe acțiuni care se execută concurent trebuie planificate astfel încît efectul total să fie acelaşi ca și cum ele s-ar fi executat serial.

2. ***Atomicitate*** O acțiune este fie executată fară nici o eroare în decursul vreunei operații, fie nu are nici un efect.

3. ***Permanența*** Efectul unei acțiuni executată cu succes nu poate fi pierdut, cu exceptia cazului erorilor catastrofale.

O acțiune poate genera un număr oarecare de apeluri de operații asupra mai multor obiecte. Ea se consideră executată cu succes numai dacă toate operațiile au fost executate fără erori. O acțiune terminată cu succes este denumită *desavîrsită* ("*commits*"); o acțiune care nu poate fi terminată cu succes datorită unei erori aparute într-o operație se spune că s-a terminat prin abort. Dacă o acțiune se termina prin *commit*, modificările efectuate asupra obiectelor sunt făcute permanente prin înregistrarea lor în memoria secundară; în caz contrar toate aceste modificari sunt inversate, astfel încît obiectul respectiv ramîne în starea în care era înaintea acțiunii.

Cel mai cunoscut mecanism pentru desăvîrșirea unei acțiuni este protocolul *two-phase committ:* [Tan88][Gos91][Cri88][Lam81](vezi 7.3.) :

(1) O cerere ***"prepare"*** este transmisă tuturor obiectelor afectate;

(2) Un obiect care a recepționat o cerere ***"prepare"*** scrie modificările asupra obiectului în memoria secundară și returnează un mesaj de confirmare;

(3) Dacă toate obiectele implicate în acțiune au returnat mesaje de confirmare, se transmite un mesaj ***"commit"*** către toate obiectele. În caz contrar, se transmite un mesaj ***"abort"*** către toate obiectele și dialogul este terminat.

(4) Un obiect care a primit mesajul "***commit***" permanentizează modificările asupra obiectelor, eliberează structurile de control (eventual lock-uri, de exemplu) și returnează un al doilea mesaj de confirmare. Un obiect care a primit mesajul ***"abort"*** inversează modificările și eliberează structurile de control.

(5) O cerere *commit* neconfirmată este repetată pîna la primirea confirmarii. Cînd toate confirmările au fost primite, protocolul two-phase commit se consideră încheiat.

Inițierea protocolului de desăvîrșire a unei acțiuni (sau de execuție *commit*) poate fi în responsabilitatea sistemului SPDBO sau a utilizatorilor. Cea mai simplă soluție este așa numitul ***mecanismul cererii (request)***, în care utilizatorul este cel care decide cînd trebuie inițiată procedura de "*commit*". Un utilizator nu este obligat să termine fiecare acțiune prin *commit*; acest lucru poate îmbunătăți performanțele sistemului, dar se pierde caracteristica de atomicitate și permanență.

Pentru a garanta toate proprietățile unei acțiuni, sistemul SPDBO trebuie să fie cel responsabil pentru gestiunea acțiunilor și desăvîrșirea lor cînd se pot încheia cu succes.

În ***mecanismul tranzacției***, cînd toate operațiile unei acțiuni au fost executate cu succes, sistemul este cel care realizează procedura de *commit* pentru a permanentiza toate modificările. Dacă cel puțin o operație se termină prin eroare, tranzacția trebuie terminată prin abort.

O extensie a mecanismului tranzacțiilor este mecanismul ***tranzacțiilor încuibate***. Conform acestui mecanism, un singur nivel exterior de acțiuni crează și gestionează mai multe subnivele de ***subacțiuni***. Eșuarea unei subacțiuni nu conduce obligatoriu la esuarea intregii acțiuni; acțiunea poate alege orice decizie în urma esuării unei subacțiuni. De exemplu poate alege repetarea execuției subacțiunii, poate termina prin abort întreaga acțiune sau poate ignora pur și simplu terminarea eronată a subacțiunii. Permanentizarea modificarilor facute de subacțiuni se face numai cînd întreaga acțiune execută "*commit*". Dacă acțiunea se termina cu abort toate modificările facute de subacțiuni sunt inversate. Mecanismul tranzacțiilor incuibate are avantajul unui control mai detaliat al unei acțiuni permițînd eșuarea subacțiunilor, dar are dezavantajul unei reprezentări mai complicate in memorie a obiectelor pentru a permite inversarea efectelor unei subacțiuni eșuate.

6.3.2. Sincronizarea acțiunilor

O importantă funcție a unui SPDBO este interdicția conflictelor și interferențelor acțiunilor la nivelul aceluiași obiect. O acțiune nu trebuie să poată observa sau modifica starea unui obiect care a fost parțial modificat de o altă acțiune care nu este terminată. În caz contrar, se poate ajunge la situația de *abort în cascadă*: orice acțiune care a observat starea parțial modificată a unui obiect de către o altă acțiune care mai tîrziu a fost terminată cu abort trebuie la rîndul ei terminată cu abort. Pentru a elimina situațiile de abort în cascadă și, pe de altă parte, pentru a pastra consistența obiectelor este necesară o *schema de sincronizare*.

Schemele de sincronizare cunoscute se clasifică în scheme ***pesimiste*** și ***optimiste***. În cadrul schemelor de sincronizare ***pesimiste***, sistemul trebuie să ia masurile potrivite pentru a preveni conflictele de acces. O acțiune care apelează o operație asupra unui obiect asupra caruia se execută o altă acțiune trebuie suspendată pîna la terminarea

BUPT

acesteia sau terminată anormal. Cel mai cunoscut mecanism de sincronizare dintre cele înglobate în schemele pesimiste este cele bazate pe lock-uri Read/Write (vezi 7.4.4.); de asemenea mai sunt utilizate semafoare (eventual semafoare distribuite, vezi 4.3.4.1.) şi mecanisme bazate pe mărcile de timp (vezi 4.3.1.2.).

În cadrul schemelor de sincronizare ***optimiste***, nu se ia nici o masură la nivelul unui obiect în cadrul caruia se execută concurent mai multe acţiuni. În schimb, la comiterea unei acţiuni se testează condiţiile de seriabilitate astfel încît informaţiile de stare luate în considerare de acţiune să nu intre în conflict cu modificările realizate de orice altă acţiune care s-a terminat. Dacă informaţiile observate de o acţiune nu au fost alterate de o altă acţiune care s-a terminat prin *commit*, atunci acţiunea poate fi *desavîrsită (commit)*; în caz contrar, modificările realizate de acţiunea respectivă, bazate pe informaţiile observate anterior vor fi incorecte şi în consecinţă acţiunea trebuie terminată prin *abort*.

Schemele de sincronizare optimiste permit cel mai mare grad de concurenţă posibil la nivelul unui obiect; acţiunile nu sunt niciodată puse în aşteptare, ca în schemele pesimiste. Poblema majoră a schemelor optimiste este însă faptul că acţiuni executate fără erori pot fi terminate prin abort; în plus trebuie menţinute în memorie mai multe copii ale obiectului, iar în memoria secundară înregistrate toate modificările realizate de aceste acţiuni.

O schema pesimistă evită regia de sistem sporită datorată reexecuţiei cererilor şi inversării efectuării modificarilor unei acţiuni pe seama reducerii concurenţei. De exemplu, două acţiuni care examinează şi modifică părţi distincte ale unui aceluiasi obiect nu pot fi executate concurent, desi nu există nici un conflict de informaţii. O schema pesimistă, pe de altă parte, evită regia de sistem datorată întîrzierii execuţiei operaţiilor asupra unui obiect pe seama execuţiei unor operaţii de inversare a efectului modificarilor ("***undo***") sau a instalării modificărilor ("***redo***").

De aceea, o schema pesimistă se comporta mai bine decît o schema optimistă cînd conflictele sunt frecvente pe cînd, în cazul contrar, cînd conflictele sunt mai puţin frecvente, o schema optimistă se comportă mai bine decît o alta pesimistă.

6.3.3. Siguranţa obiectelor

Un SPDBO trebuie să fie capabil să detecteze şi recupereze obiectele în cazul de defect al unui nod. Aceasta este o caracteristică importantă deoarece cu cît numărul de componente într-un sistem creşte cu atît creşte şi probabilitatea aparitiei unui defect. Există două metode generale de creştere aa fiabilităţi unui SPDBO: o prima metoda propune recuperarea unui obiect imediat dupa detectarea şi înlăturarea defectului, limitînd la maximum timpul în care obiectul ramîne inactiv; a doua metodă propune replicarea obiectelor pe mai multe noduri.

Metoda recuperării obiectelor are două variante: varianta recuperării ***roll-back*** şi varianta recuperării ***roll-forward***.

În varianta recuperării ***roll-back*** un obiect defect este restaurat la ultima sa stare consistentă înregistrată în memoria secundară de către procedura *commit*. Toate acţiunile în curs de desfasurare în momentul apariţiei defectului vor fi fără efect (vor fi pierdute).

În varianta ***recuperării roll-forward*** un obiect defect, ***precum şi toate obiectele cu care acesta a lucrat*** sunt restaurate la ultima lor stare consistentă înregistrată în memoria secundară de procedura *commit*. Toate procesele şi acţiunile în progres în momentul apariţiei defectului sunt restartate şi executate complet. Deoarece în aceasta variantă de recuperare, sistemul trebuie să apară dupa recuperare ca şi cînd nu ar fi aparut nici un defect, varianta este mai complexă decît cea a recuperării roll-back; în plus are dezavantajul că dacă eroarea este software ea va apare din nou cînd obiectul este restaurat.

Metoda obiectelor replicate permite existenţa mai multor copii ale unui obiect pe mai multe noduri. Aceasta permite unui SPDBO să tolereze defecte la un anumit număr de noduri menţinînd totuşi funcţionalitatea sistemului în ansamblu. Metoda ridică probleme legate de mentinerea consistenţei informaţiilor între copiile unui obiect şi de sincronizarea activităţii clienţilor.

Cea mai simplă strategie este de a permite numai ***obiectelor imuabile*** să fie replicate. Obiectele imuabile sunt cele care pot fi numai examinate de clienţi şi deci a căror stare nu poate fi modificată. Acest tip de obiecte pot fi replicate fără probleme de menţinerea consistenţei stării şi a sincronizării accesului.

O alternativă la strategia de mai sus este strategia ***copiei primare***. O copie a obiectului este desemnată drept copia primară, in timp ce restul copiilor sunt ordonate şi menţinute în staţii separate drept copii secundare. Cererile de citire pot fi direcţionate către orice copie, în schimb cererile de scriere nu pot fi direcţionate decît către copia primară care propagă apoi toate modificările şi către copiile secundare. Există două variante ale strategiei copiei primare: ***varianta copiei primare statice*** şi ***varianta copiei primare dinamice***. În varianta copiei primare statice, dacă serviciul copiei primare devine indisponibil, cererile de scriere nu sunt servite decît la recuperarea copiei primare. În varianta copiei primare dinamice, dacă serviciul copiei primare devine indisponibil, sistemul desemnează una din copiile secundare drept copie primară, putîndu-se astfel trata în continuare cererile de modificare ale obiectului; problema care apare este ca identitatea copiei primare precum şi resursele asociate ei precum porturi şi lock-uri să poată fi asociate copiei secundare desemnate ca primare. O altă strategie este ***strategia copiilor egale*** în care nu se desemnează un obiect primar sau secundar; fiecare copie a obiectului se considera egală cu oricare alta. Atît cererile de citiere cît şi cele de scriere pot fi servite de oricare copie; în acest caz însă este necesară cooperarea unora dintre copii în scopul servirii corecte a acestor cereri. Există mai multe variante la aceasta strategie: *sistemul votului*, ***al copiilor disponibile*** sau *sistemul regenerarii*.

BUPT

6.4. Interacţiunea obiectelor

Un SPDBO este responsabil şi pentru gestionarea interacţiunii obiectelor. Atunci cînd o acţiune apelează o operaţie asupra unui obiect, sistemul trebuie să localizeze obiectul respectiv, să transmită cererea către acesta şi eventual să returneze raspunsul.

6.4.1. Localizarea obiectelor

Un SPDBO trebuie să furnizeze utilizatorilor sai caracteristica de ***transparenţă*** în localizarea obiectelor, astfel încît un client să nu fie obligat să cunoască adresa fizică a nodului în care un obiect este rezident; la orice cerere de execuţie a unei operaţii sistemul trebuie să identifice ce obiect este implicat şi unde este rezident. Consecinţa este că sistemul trebuie să asigneze cîte un identificator fiecărui obiect, iar aceşti identificatori trebuie să fie unici. Mecanismul de localizare poate chiar să fie atît de flexibil încît să permită migrarea obiectelor de la un nod la altul.

O primă soluţie a problemelor de mai sus este ***codificarea localizării obiectului*** chiar în identificatorul obiectului. În momentul apelului, sistemul examinează cîmpul respectiv din identificatorul obiectului şi decodifică adresa nodului în care obiectul este rezident. Soluţia este foarte simplă, eficientă, dar şi restrictivă: odată ce un obiect a fost asignat unui nod el nu mai poate fi mutat, deoarece identificatorul obiectului trebuie modificat in acest caz. În consecinţă, pe toată durata de viaţă a sa, un obiect este fixat pe un nod.

O altă soluţie este strategia ***serverului de nume distribuit***. Conform acestei strategii, sistemul crează un grup de servere de nume ale obiectelor, menţinute în general, (dar nu şi totdeauna necesar) pe toate nodurile. Aceste servere cooperează astfel încît, per total ele conţin toate informaţiile necesare pentru localizarea oricărui obiect din sistem. Există două variante ale acestei strategii. În prima variantă, un server de nume menţine informaţii complete pentru localizarea oricărui obiect astfel încît orice server poate deservi orice cerere de localizare. În cea de a doua variantă, un server de nume menţine informaţii parţiale pentru localizare; dacă o cerere de localizare nu poate fi rezolvată de un server de nume se delegă altul s.a.m.d. În aceasta variantă, cel puţin un server trebuie înştiinţat atunci cînd un obiect este creat.

O altă strategie este strategia ***cache/broadcast***. În fiecare nodie se menţine o memorie cache în care se memorează localizarea ultimelor "n" obiecte referite. La orice referire a unei cereri de operaţii asupra unui obiect se examinează aceasta memorie cache; dacă localizarea obiectului nu este gasită în aceasta lista, se trimite un mesaj broadcast în reţea pentru a se determina nodul care conţine obiectul respectiv. Fiecare nod care primeşte acest mesaj, caută în propria sa memorie cache pentru a vedea dacă deţine acest obiect; în caz afirmativ trimite un mesaj de confirmare la nodul emiţător, care îşi actualizează memoria cache relativ la acest obiect. Strategia este foarte eficientă, deoarece este de aşteptat ca localizarea obiectelor să fie gasită în majoritatea cazurilor în memoria cache locala. Este, deasemenea, flexibilă deoarece permite migrarea unui obiect, fără a fi nevoie de transmisie de mesaje la alte noduri sau la un server de nume. Dezavantajul este faptul că trebuie transmise mesaje broadcast, ceea ce înseamnă o încarcare mare a reţelei; avînd drept efect conturbarea funcţionării mai multor noduri deşi unul singur este implicat în dialog.

6.4.2. Apelul de operaţii

Atunci cînd un client apelează o operaţie asupra unui obiect, sistemul SPDBO este cel care trebuie să transmită cererea către serverul obiectului respectiv şi să returneze raspunsul înapoi la apelant. Există două strategii pentru gestionarea acestor cereri: ***strategia transmiterii de mesaje*** şi ***strategia apelului direct***.

Prima strategie, cea a ***transmiterii de mesaje*** este, în mod tipic folosită în sistemele care implementează modelul obiectului activ. Cînd un client apelează o operaţie asupra unui obiect, parametrii cererii sunt împachetaţi în mesaj; mesajul este transmis procesului server sau la un port asociat obiectului respectiv. Procesul server recepţionează mesajul, despachetează parametrii din mesaj, execută operaţia şi împachetează rezultatul într-un alt mesaj pe care-l transmite apelantului. Legatura client-server este dinamică şi se realizează la fiecare apel. Un dezavantaj este complexitatea dialogului pentru apeluri in acelaşi nod.

Cea de-a doua strategie, ***a apelului direct*** este proprie sistemelor care implementează modelul obiectului pasiv, in care un singur proces este responsabil pentru execuţia tuturor operaţiilor asociate unei acţiuni. Consecinta este ca un proces poate migra în funcţie de operaţie de la un obiect la alt obiect.

Atunci cînd un proces apelează o operaţie asupra unui obiect rezident pe aceeaşi statie, execută urmatoarele patru etape(figura 6.6.) [CC91]:

(1) Starea procesului şi obiectele apelate se inregistrează în zona stiva a procesului. Sistemul poate proteja aceste zone impotriva examinării sau alterării ei de alte apeluri.

(2) Parametrii apelului sunt adăugaţi în stivă.

(3) Obiectul apelat este incarcat în memorie şi se execută un apel de procedură pentru operaţia cerută.

(4) La terminarea operaţiei, rezultatele sunt returnate clientului şi procesul este în starea avuta înaintea apelului.

Atunci cînd un proces execută o acţiune asupra unui obiect rezident în alt nod, se execută în plus următoarele trei etape, primele două dupa (1), iar a treia dupa (3) (fig 6.7.):

(1.1) Se formează un mesaj cuprinzînd parametrii cererii care se trimite nodului în care obiectul este rezident

BUPT

(1.2) Nodul care primeste mesajul crează un proces care se execută în continuarea procesului curent

(3.1) Cînd operaţia a fost executată, se formează un mesaj conţinînd rezultatele care se transmite nodului clientului. Procesul din nodul de la distanţă este apoi distrus.

Apelul direct apare astfel asemănător unui apel de procedură la distanţă. Metoda are dezavantajul creerii şi distrugerii de noi procese în nodurile în care obiectele sunt rezidente.

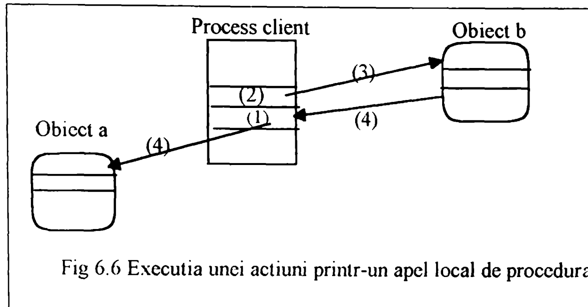


Fig 6.6 Executia unei actiuni printr-un apel local de procedura

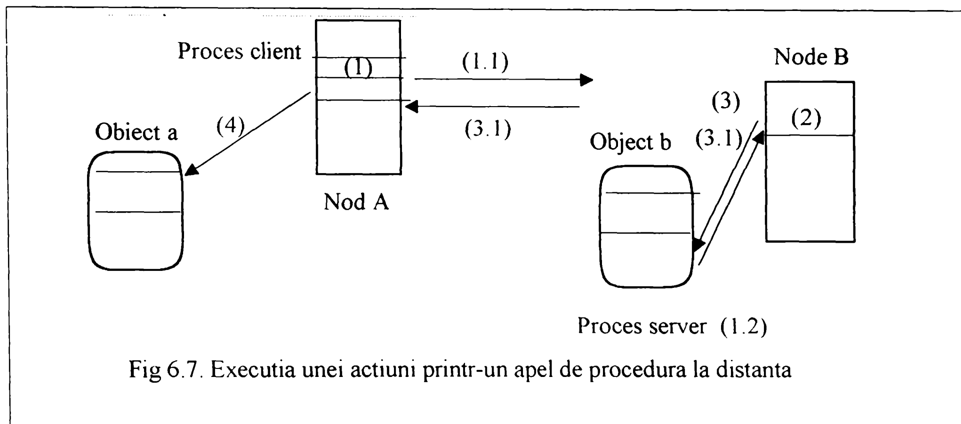


Fig 6.7. Executia unei actiuni printr-un apel de procedura la distanta

6.4.3. Detectarea eşecurilor în apelarea operaţiilor

Eşecurile în apelul unei operaţii se clasifică în ***eşecuri permanente*** şi ***eşecuri tranzitorii***. Un *eşec permanent* se defineşte ca un eşec apărut înainte de începerea execuţiei operaţiei. Cea mai cunoscută situaţie de acest gen este cea în care obiectul apelat nu poate fi localizat. Aceste tipuri de eşecuri sunt uşor de detectat şi tratat.

Un ***eşec tranzitoriu*** este un eşec care apare în timp ce apelul operaţiei se execută. Un astfel de eşec apare deci dupa acceptarea unei operaţii, dar înainte ca modificările facute să devină permanente prin desăvîrşirea acţiunii (înainte de terminarea prin "*commit*"). Aceste eşecuri sunt mai greu de detectat şi de tratat deoarece pot avea cauze diverse. De exemplu, un eşec al unui apel de operaţie se poate datora eşecului obiectului client, obiectului server sau liniilor de comunicaţie. Un SPDBO trebuie să prevadă mecanisme pentru detectarea eşecurilor atît la nivelul obiectului client cit şi la cel al serverului precum şi pentru recuperare după astfel de eşecuri.

Sunt cunoscute mai multe metode pentru detectarea unor astfel de eşecuri: bazate pe mecanismul time-out, pe teste ale obiectului sau ale apelurilor. Dacă un eşec tranzitoriu nu este detectat de client, clientul şi acţiunea corespunzătoare pot fi blocaţi la nesfîrşit. În consecinţă, un client trebuie să poată detecta eşecul unui apel de operaţie şi să poată iniţia o procedură de recuperare a eşecului. Procedura de recuperare trebuie să elibereze resursele deţinute de client şi să termine acţiunea cu eroare, eventual să repete apelul operaţiei.

Dacă un eşec tranzitoriu nu este detectat de obiectul server, resurse ale sistemului pot fi irosite nedefinit. Un obiect startat în urma unui apel de operaţie, ale cărei rezultate însă nu mai sunt preluate, este denumit *orfan*; el poate fi generat datorită unui eşec al unui obiect client, datorită unei tranzacţii terminate cu abort, sau datorită unei erori ale reţelei şi iroseşte resurse precum lock-uri şi timp procesor. În consecinţă, trebuie eliminat cît mai repede posibil (vezi 3.4.5. pentru metode de eliminare a orfanilor).

BUPT

6.5. Reprezentarea obiectelor în memorie

Un SPDBO trebuie să furnizeze mecanisme pentru gestionarea resurselor precum memoria principală, dispozitive de memorie externă, noduri ale reţelei. Specific SPDBO este modul de reprezentare în memoria internă şi secundară a obiectelor, precum şi modul de transfer al obiectelor între cele două, ca şi cum se asignează obiectele procesoarelor

Obiectele a căror reprezentare este pierduta în urma defectării nodului se numesc ***obiecte volatile***. Obiectele volatile sunt temporare, sunt rezidente numai în memoria internă şi deci uşor de manipulat. Obiectele a căror reprezentare se menţine după defectarea nodului, în limitele unei mari probabilităţi, se numesc ***obiecte persistente***. Reprezentarea obiectelor persistente trebuie făcută obligatoriu şi în memoria secundară; una sau chiar mai multe copii ale acestui obiect se poate găsi şi în memoria internă. Acţiunile modifică versiunea din memorie a unui obiect persistent. Versiunea din memoria secundară este modificată numai după ce acţiunea se termina prin *commit*, asigurîndu-se astfel menţinerea unei versiuni consistente şi stabile a obiectului, însă cu preţul regiei de sistem sporite prin realizarea operaţiilor de copiere în memoria secundară.

Un obiect persistent rezident atît în memoria internă cît şi în memoria secundară este numit ***obiect activ***. Un obiect persistent avînd reprezentarea numai în memoria secundară este numit ***obiect inactiv***. La orice apel al unei operaţii asupra unui obiect inactiv se copiaza o copie a obiectului în memoria internă; eventual se pot crea şi noi procese pentru gestionarea acestei copii volatile. Dacă procedura de *commit* relativă la obiect se termină cu succes, obiectul este copiat în memoria secundară (actualizat), iar memoria internă ocupată de copia sa dealocată. Aceasta salvare automată a obiectelor active persistente în memoria secundară şi încarcarea lor din memoria secundară trebuie realizată transparent de SPDBO. Acesta trebuie să ascundă faptul că se foloseşte memorie secundară pentru memorarea obiectelor, în aparenţă fiind vorba numai de obiecte întotdeauna disponibile. Modul în care un obiect este reprezentat în memorie depinde de schema de sincronizare folosită şi de modul de gestionare a acţiunilor prevazute în SPDBO. Schema de sincronizare influentează numărul de versiuni ale obiectului, iar modul de gestionare al acţiunilor influentează reprezentarea fiecărei versiuni.

Dacă sistemul SPDBO utilizează o schemă de sincronizare pesimistă, se menţine în memorie, în general o singură versiune a obiectului. Toate acţiunile care apelează operaţii asupra unui acelaşi obiect modifică aceeaşi versiune volatilă. Dacă schema de sincronizare este optimistă se menţin în memorie noi versiuni ale fiecarui obiect creat. Fiecărei acţiuni care apelează un obiect i se asignează propria versiune volatilă a obiectului asupra căreia operează toate modificarile. Astfel se permite ca multiple acţiuni să apeleze acelaşi obiect simultan fără interferenţe.

Reprezentarea fiecarui obiect în memorie depinde de prezenţa sau absenţa facilităţii de tranzacţii încuibate. Dacă nu se utilizează tranzacţii încuibate, este suficientă reprezentarea obiectului cu o singură copie în memorie. Dacă tranzacţia eşuează, este suficientă dezalocarea copiei respective. În cazul cînd sunt permise tranzacţii încuibate, lucrurile se complică deoarece mai multe subacţiuni pot modifică acelaşi obiect. Modificările asupra obiectului nu sunt permanentizate decît cînd acţiunea de la nivelul cel mai superior execută *committ* şi, pe de alta parte, fiecare subacţiune poate fi terminată cu *commit* sau poate eşua. Dacă o subacţiune modifică un obiect şi apoi se termină prin abort, modificările trebuie invalidate ("undone") astfel încît obiectul să poată fi restaurat la starea de dinaintea execuţiei subacţiunii. În consecinţă, este necesar un mecanism suplimentar pentru a se putea inversa efectul modificarilor unei subacţiuni. Sunt cunoscute două procedee: utilizarea unui jurnal undo/redo sau menţinerea unei copii "hotstandby", o versiune a obiectului creat de subacţiune.

Un sistem SPDBO trebuie să menţină în memoria secundară suficiente informaţii pentru ca un obiect persistent să poată fi restaurat la o stare consistentă, ca de exemplu dupa un defect al nodului în care obiectul este rezident. Un sistem SPDBO poate înregistra întreaga stare a unui obiect în momentul execuţiei *commit* sau poate înregistra numai modificările efectuate asupra obiectului, relative la o stare inregistrată anterior.

În prima metoda, denumită ***metoda punctelor de reluare*** (***checkpoint scheme***) se înregistrează în memoria secundară întreaga stare a obiectului modificat, cînd acţiunea respectivă execută *commit*. În faza de pre-*commit* a acţiunii versiunea modificată este scrisă în memoria secundară, fără interferenţa cu versiunea veche a obiectului. În faza de *commit*, versiunea modificată înlocuieşte versiunea veche care este apoi dezalocată. Dacă se execută însă abort, se dezalocă versiunea modificată în timp ce versiunea veche rămîne nemodificată.

Un avantaj al acestei metode este faptul că menţine în memoria secundară numai o singură copie a obiectului. Dezavantajul este că şi în cazul unor puţine modificări asupra unui obiect avînd informaţii de stare de dimensiuni mari trebuie înregistrate integral în memoria secundară.

O variantă a acestei metode este ***metoda istoriei punctelor de reluare***. *Metoda* propune reprezentarea unui obiect persistent în memoria secundară ca o colecţie ordonată de puncte de reluare. În momentul în care trebuie creată o nouă versiune a unui obiect, vechea versiune nu este distrusa; noua versiune se adauga secvenţei de puncte de reluare. În afară de dezavantajul spaţiului aditional necesar în memoria secundară, metoda mai are şi dezavantajul problemei determinării momentului în care vechile puncte de reluare nu mai sunt necesare şi, de aceea şterse.

O metoda complet diferită este cea a *jurnalelor log*. Metoda propune ca numai modificările executate asupra unui obiect persistent să fie inregistrate într-un jurnal menţinut în memoria secundară (vezi 7.3.4.). Cantitatea de informaţie inregistrată în memoria secundară depinde de schema particulară de inregistrare în jurnal folosită. Evident că, faţă de metoda precedentă, metoda jurnalelor log foloseşte spaţiul memoriei secundare mai eficient. O consecinţă a memorării numai a modificarilor este faptul că, cu cît jurnalul log creşte, cu atît creşte şi regia de sistem necesară menţinerii lui ca şi pentru tratarea recuperarilor.

BUPT

Capitolul 7

CONTROLUL CONCURENŢEI, INTERBLOCĂRI SI RECUPERAEA OBIECTELOR

Acest capitol tratează în principal al doilea aspect al sincronizării în sisteme distribuite, prezentat în introducerea capitolului 4. Alaturi însă de problemele de control al comunicației și de tranzacțiile atomice sunt tratate și alte probleme legate de acestea, precum recuperarea tranzacțiilor și controlul interblocărilor.

7.1 Introducere

Se apreciază că proiecterea aplicațiilor distribuite fiabile poate fi obținută prin două metode generale, conceptual diferite: (1) prima metodă utilizează conceptul de *tranzacție atomica*; (2) a doua metodă utilizează copii multiple ale obiectelor necesare aplicațiilor, soluționînd corect problema actualizării copiilor ([Gos91], [Tan93], [Tan96], [CC91]). Problemele în legatură cu controlul concurenței sunt legate în sistemele distribuite de problemele de comunicație; s-a aratat în capitolul 4 că sincronizarea în sistemele ditribuite nu se poate realiza fără comunicație.

Controlul concurenței este și parte componentă a protocolului OSI CCR (*Commitment, Concurrency and Recovery*) care guvernează desfășurarea tranzacțiilor atomice. Controlul concurenței se consideră un aspect comun al sistemelor de operare distribuite și bazelor de date distribuite. Conceptul de tranzacție a apărut inițial în bazele de date, ca un instrument în menținerea consistentei datelor; consistența trebuia menținută atît în cazuri de avarii ale unor noduri, cît și în cazul acceselor concurente. Bazele de date nu sunt însă singurele aplicații în care trebuie menținută consistența datelor, în situațiile critice de mai sus; majoritatea sistemelor de fișiere pentru rețele de calculatoare de astăzi oferă facilități pentru blocarea la nivel de înregistrare fizică, blocare la nivel de înregistrare logică sau la nivel de fișier și, în special facilități pentru utilizarea tranzacțiilor atomice. Un exemplu este sistemul de gestiune fișiere Novell Netware care pune la dispoziția utilizatorilor API-uri de sincronizare (*Synchronization Services*) între care figurează apeluri pentru manipularea lock-urilor și pentru lucrul cu tranzacții atomice.

Un alt exemplu este sistemul distribuit de fișiere QuickSilver Distributed File System (QDFS) ([CPS93]). Serverele pentru fișiere ale acestui sistem furnizează accesul la fișiere într-un spațiu de nume structurat ca un arbore și prezintă o interfață similară cu cea din UNIX, cu excepția faptului că nu permite accesul aleatoriu la fișiere. Spre deosebire de UNIX, toate operațiile implementate în DFS sunt atomice. Aceasta înseamnă că, de exemplu, un client poate șterge, rescrie un fișier sau redenumi un catalog și apoi decide dacă modificările vor fi permanentizate sau nu. QDFS garantează că, fie toate modificările vor fi permanentizate, fie niciuna nu va fi realizată; aceasta garanție se face chiar pentru situații de avarii în rețea (defecțiuni ale nodurilor, ale liniilor de comunicație).

Tranzacțiile atomice au fost incluse de asemenea în Argus (limbajul CLU) ([Lis88], [LCJ87], [Lis90], [CC91][TCM94]), în sistemul de operare distribuit Clouds ([For88][CC91], [TCM94], [Tan96]), în TABS și Camelot ([Spe90], [SDE85], [SBD86], [CC91], [TCM94]), în Eden ([ABN85], [CC91]), în Arjuna ([CC91]), Encina (Transarc Corporation, 1991, [CC91]), Tuxedo (UNIX System Laboratories, 1991,[CC91]). Se apreciază că, la ora actuală, deși suportul tranzacțiilor atomice migrează de la bazele de date la sistemele de operare, acest lucru nu este încă unanim acceptat ([CPS93], [KS89]).

Pe de altă parte, trebuie precizat că modelul tranzacțiilor clasice, adoptat în bazele de date nu satisface cerințele unor aplicații complexe, precum aplicații CAD/CAM, medii de dezvoltare de software (SDT-uri), gestiunea rețelelor, informatică medicală, aplicații care sunt orientate pe date masive, în sensul că generează și manipulează volume mari de date. Este de dorit ca pentru aceste tipuri de aplicații să existe facilități similare cu cele traditionale în bazele de date precum adăugarea, ștergerea, regăsirea datelor, alaturi de menținerea consistenței informatiilor stocate, cu observația că mecanismul tradițional al tranzacțiilor nu este adecvat. Astfel, aplicații precum gestiunea rețelelor și informatică medicală pot necesita prelucrări în timp real; altele, precum proiectare CAD/CAM sau medii de dezvoltare software necesită cooperare între mai mulți utilizatori. Controlul concurenței în manieră conventională nu este aplicabil pentru aceste cazuri, pentru care se utilizează termenii de *tranzacții lungi* și *tranzacții cooperante*. Unii cercetători susțin chiar că termenul de tranzacție în sensul clasic își pierde semnificatia pentru tranzacțiile lungi și cele cooperante, din moment ce acestea nu mai conservă proprietățile de *atomicitate* și *seriabilitate*. Pe parcursul acestui capitol, în secțiunea 7.4.7 în special, se prezintă și cerințele specifice și mai ales soluții legate de controlul adecvat al concurenței pentru astfel de aplicații, denumite *aplicații complexe*.

BUPT

7.2 Modelul tranzacțional

În cele ce urmează, se va utiliza noțiunea de *sistem de obiecte.*
Definiție.
Se numește *sistem de obiecte* o colecție de obiecte (de exemplu înregistrări sau fișiere) vazută ca o resursă unică, asupra căreia se execută operațiile de bază, denumite *actiuni, read* și *write.*

Prelucrările asupra sistemului de obiecte pot fi văzute ca secvențe de actiuni aplicate unui subset de obiecte ale sistemului de obiecte.

7.2.1 Consistența obiectelor. Noțiunea de tranzacție

Fiecare sistem de obiecte trebuie să satisfacă un set de restricții denumite *constrangeri de consistenta.* Exemple clasice sunt cel ale unei baze de date bancare, la care o constrîngere este aceea că balanța fiecărui cont trebuie să fie egală cu suma depunerilor minus suma extragerilor și cel al unui sistem de rezervare de locuri la care o constrîngere este faptul că fiecare loc poate fi rezervat numai de un singur pasager. Starea bazei de obiecte este *consistentă* dacă sunt respectate toate constrîngerile de consistență. Uneori, constrîngerile de consistență pot fi foarte complexe. Pentru a trata și rezolva problemele de consistență a fost introdus conceptul de *tranzacție.*
Definiție.
O *tranzacție* este o secvență de actiuni, care constituie unitatea logică de lucru cu sistemul de obiecte și care transferă sistemul de obiecte dintr-o stare consistentă în altă stare consistentă.

În bazele de date tradiționale, tranzacțiile servesc pentru trei scopuri: (1) sunt unități logice care grupează acțiunile unei prelucrări; (2) sunt unități atomice a căror execuție conserva consistența bazei de date; (3) sunt unități de recuperare.

```
Balanta initiala a conturilor
A   800 000
B   400 000
C   200 000

Tranzactii
T1 ::
    BeginTransaction
     read cont A
     read cont B
    write balanta A - 100 000 in contul A
    write balanta B + 100 000 in contul B
    EndTransaction
T2::
    BeginTransaction
     read cont B
     read cont C
    write balanta B - 200 000 in contul B
    write balanta C + 200 000 in contul C
    EndTransaction
T3..
    BeginTransaction
     read cont A
     read cont B
     print balanta A
     print balanta B
     EndTransaction
```

Fig. 7.2.1.1 Tranzactii concurente

Pentru a crește performantele, este indicat să se permită execuția concurentă a tranzacțiilor. Totuși, intermixarea arbitrară a acțiunilor tranzacțiilor concurente are deseori ca rezultat o stare inconsistentă; în consecință, este necesar controlul acceselor concurente, sau pe scurt *controlul concurenței.* Există două situații de anomalie care generează stările de inconsistență: *anomalia actualizarilor pierdute,* și *anomalia regasirilor inconsistente.* Exemplul din fig. 7.2.1.1 ilustrează aceste situații.
Dacă tranzacțiile T_1 și T_2 din acest exemplu ar fi executate concurent, prin intermixarea acțiunilor s-ar putea obține o balanța finală a contului B de 500000 sau 200000; aceasta este anomalia *actualizarilor pierdute.* Dacă tranzacția T_3 citește conturile A și B în timpul execuției tranzacției T_1, atunci afișarea rezultatelor poate conduce la situația în care o sumă de bani transferată dintr-un cont în altul dispare; aceasta este situația de *regasire inconsistentă* (A este debitor dar B nu a fost creditat). Deși T_1 și T_2, individual, pastrează consistența datelor, ele prezintă însă următoarele caracteristici: (1)*inconsistența temporara*: dupa prima operație *write* fie a lui T_1, fie a lui T_2 sistemul de obiecte este inconsistent; (2) *conflict*: dacă tranzacția T_2 se execută între prima și a doua operație *write* a lui T_1, atunci suma finală a balanței conturilor va fi de 1 600 000 și nu de 1 400 000, contrazicînd cerințele de consistență.

O tranzacție poate fi terminată *normal* sau *anormal.* În primul caz se spune că este *realizată* sau *comisa*; în cel de-al doilea caz că este *abortată.* Tranzacția însăși poate cere să fie abortata, ca urmare, de exemplu, a unor date de intrare eronate; pe de altă parte, o tranzacție poate fi însă abortată și de sistem, ca urmare a violării unor constringeri de consistenta.

Pentru a se respecta constrîngerile de consistență impuse unui sistem de obiecte, fără a se impune condiții excesiv de restrictive asupra prelucrărilor concurente ale aplicațiilor, o tranzacție trebuie să aibă patru proprietăți cunoscute sub numele de *proprietăți ACID* în literatură ([CPS93], [BHG87], [SDE85]): (1) Proprietatea de *atomicitate* este proprietatea conform căreia o tranzacție fie că se realizează, caz în care toate actualizările asupra obiectelor se permanentizează, fie este abortată, caz în care

efectul ei este nul asupra obiectelor. (2) Proprietatea de *consistență* este proprietatea conform căreia o tranzacție transformă un sistem de obiecte dintr-o stare consistentă în alta. (3) Proprietatea de *izolare* sau de *seriabilitate* este proprietatea care garantează că efectul tranzacțiilor care se execută concurent este echivalent cu efectul execuției lor în mod serial, într-o ordine oarecare. (4) Proprietatea de *durabilitate* sau de *persistență* a efectului este proprietatea conform căreia efectele unei tranzacții realizate rămîn valabile şi dupa o defecțiune a sistemului.

O tranzacție care execută operații asupra unor obiecte dispersate într-un sistem distribuit se numeşte *tranzacție distribuită*. Distribuirea obiectelor pe mai multe noduri creşte performanța sistemului legată de concurență şi incurajează utilizarea copiilor multiple pentru un obiect. Pentru a creşte şi mai mult gradul de concurență, tranzacțiile pot fi *încuibate* ([Spe84]); tranzacțiile încuibate se caracterizează prin următoarele elemente: (1) o tranzacție de nivel superior poate iniția mai multe tranzacții încuibate care se execută în paralel; acestea, la randul lor, pot crea alte tranzacții, s.a.m.d. În modelul general, o tranzacție părinte îşi suspendă activitatea pînă cînd fiii sai se termină (normal sau anormal); (2) o tranzacție fiu poate obține un *lock* (folosit de mecanismul de concurenta, vezi subcapitolul 7.4.4.) de la o tranzacție ascendenta; (3) cînd o tranzacție se termină cu *commit*, toate *lock*-urile sale, inclusiv cele moştenite, sunt returnate tranzacției părinte; astfel, numai la terminărea tranzacției celei mai exterioare, se eliberează *lock*-urile. Analog, efectul tranzacțiilor încuibate se permanentizează numai cînd tranzacția cea mai exterioară se termină (cu *commit*); (4) dacă are loc un incident hardware, ce generează abortarea unei tranzacții interioare, atunci toate *lock*-urile deținute de acea tranzacție sunt returnate tranzacției părinte, care este anunțată de terminarea anormală. Tranzacția părinte poate continua sau se poate aborta ea însăşi.

Se apreciază că tranzacțiile atomice, utilizate împreună cu mecanismul RPC, cu tehnici de autentificare şi protectie, şi cu un suport lingvistic adecvat, pot constitui o bază importantă pentru construcția de programe sigure ([Gos91]).

7.2.2 Modelul de lucru pentru sistemul distribuit de obiecte

Se prezintă în aceasta secțiune modelul de lucru al sistemului distribuit de obiecte: *modelul obiect-entitate*.

Colecțiile de obiecte sunt împrăştiate pe mai multe noduri; aceste noduri sunt interconectate într-o rețea şi identificate în mod unic printr-un număr. Nodurile comunică între ele prin mesaje, livrate cu o întîrziere oarecare în cadrul rețelei; se presupune însă că toate mesajele sunt livrate, şi, în plus, sunt livrate în ordinea în care au fost emise. Sistemul distribuit de obiecte constă într-un set de *entități* (de exemplu înregistrări sau fişiere), care primesc denumiri unice şi servesc ca unitate indivizibilă pentru acces. O entitate se materializează în unul sau mai multe obiecte de date, fiecare identificat prin perechea <nume_entitate> <identificator_nod>. Replicarea unei entități are loc atunci cînd o entitate este reprezentată de mai multe obiecte identice. Replicarea entităților poate conduce la o îmbunătățire a performantelor dacă costul memorării şi menținerii obiectelor copii este mai mic decît costul de acces; de asemenea, replicarea poate mări viteza de acces la obiecte şi siguranța de funcționare sistemului, în sensul că dacă un nod devine inaccesibil, accesul este posibil la un nod care deține o altă copie a obiectului.

În figura 7.2.2.1 se arată corespondența între entități şi obiecte.

O entitate E, replicată în nodurile S_1,... S_K este reprezentată prin obiectele $<E,S_1>$,... $<E,S_K>$. Fiecare tranzacție T este supervizată printr-un manager de tranzacție, al cărui scop principal este de a supraveghea prelucrările distribuite. Se definesc patru primitive la interfața tranzacție-manager: *read*(E_i) care returnează valoarea entității E_i, *write*(E_i, *valoare*) care setează valoarea entității E_i la valoarea argumentului şi primitivele *BeginTransaction* şi *EndTransaction* care delimitează o tranzacție. Fiecare cerere de citire a unei entități este translatată de managerul de tranzacții într-o acțiune *read* de forma: *read(T,read obiect*$<E,S>$*,V)* asupra obiectului corespunzător; analog, fiecare cerere de scriere a unei entități este translatată într-un set de acțiuni *write* de forma: *write(T,write obiect*$<E,S>$*,V)*, cîte o acțiune *write* pentru fiecare obiect care constituie o copie (dacă se utilizează replicarea entităților). În plus, pentru a comite tranzacția, managerul de tranzacții generează cîte o primitiva *commit* de forma *(T,commitobject,*$<-,S>$*)*, pentru fiecare nod S la care s-a executat o acțiune.

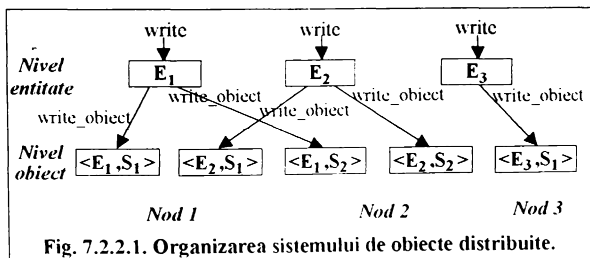


Fig. 7.2.2.1. Organizarea sistemului de obiecte distribuite.

Managerii de obiecte sunt procesele care gestionează obiectele şi execută asupra obiectelor acțiunile cerute de managerul de tranzacții. Ca raspuns la o acțiune *read*, de exemplu, managerul de obiecte returnează managerului de tranzacții valoarea obiectului. O acțiune *commit object* are ca efect permanentizarea efectelor acțiunilor *write* dintr-o tranzacție asupra acelui obiect. La un moment dat, pentru o tranzacție pot fi în lucru mai mulți manageri de obiecte, supravegheați de managerul de tranzacție la care s-a deschis tranzacția cu primitiva *BeginTransaction*. Execuția cererilor la nivelul unui manager de obiecte, pentru acelaşi obiect, se face secvențial.

În modelul prezentat se va considera că fiecare entitate fie nu este replicată, fie este complet replicată (în toate nodurile). Se consideră că această restricție nu este o restricție conceptuală majoră, deoarece o entitate parțial replicată poate fi tratată ca o grupare de entități distincte punîndu-se o constrîngere globală de consistență: aceea că toate entitățile distincte să conțina aceleaşi valori. Se impune ca o cerere de scriere pentru o entitate să genereze actualizarea fiecărei copii a unei entități, în timp ce o cerere de citire să acceseze o singură copie. Cu această manieră de a trata replicarea, modelul poate fi generalizat la modelul cu voturi al lui Gifford (vezi secțiunea 1.3.2.2). Pe

BUPT

parcursul acestui capitol, dacă nu se va specifica explicit, se va considera că o entitate nu este replicată, ceea ce implică mapare unu la unu entitate-obiect.

Există mai multe modele pentru descrierea și reprezentarea unei tranzacții într-un sistem distribuit de obiecte. Prim model este acela în care se creează cîte un *proces agent* în fiecare nod implicat în execuția unei tranzacții; procesul agent obține accesul la obiectul respectiv și va elibera obiectul dupa îndeplinirea cererii și, eventual, va returna un semnal de sfîr acces apelantului sau. În acest model (Menasce și Muntz) se consideră că procesele agent se creează cînd este necesar acces la un obiect situat la distanță și se distrug cînd accesul a fost realizat. Un al doilea model (Gray) consideră că procese de la distanță se creează la începutul tranzacției și rămîn în viață pînă la terminarea ei. Un al treilea model (Rosenkran consideră că o tranzacție originară dintr-un nod migrează dintr-un nod în altul, dupa cum se generează acțiunile asu obiectelor. În cele ce urmează în acest capitol se va considera primul model.

Există două modele de arhitectură în ceea ce privește managerii de tranzacții și managerii de obiecte: *model integrat* și *modelul client-server*. În *modelul integrat* (fig.7.2.2.2), toate nodurile sunt dotate cu manageri de tranzacție eventual, cu managerii de obiecte dacă există sisteme de obiecte pe acele noduri. Modelul poate fi folosit în rețele locale dar în rețele de larga raspîndire geografica. În *modelul client-server* (fig.7.2.2.3), gestiunea tranzacțiilor (clienti) este separată gestiunea obiectelor (servere). Acest model este utilizat numai în rețelele locale.

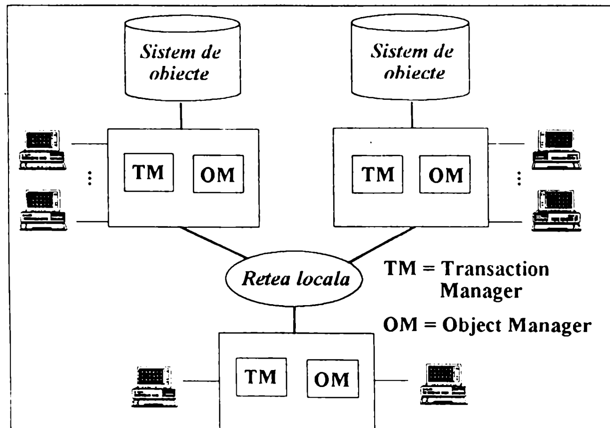


Fig. 7.2.2.2. Modelul integrat.

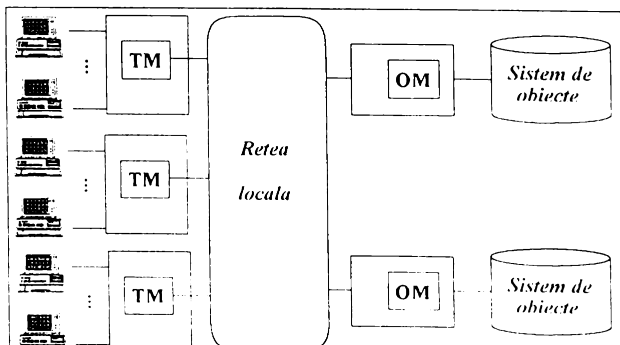


Fig. 7.2.2.3. Modelul client-server.

BUPT

7.2.3 Cerințele modelului sistemului distribuit de obiecte

Un sistem distribuit de obiecte trebuie să îndeplinească următoarele *cerințe*: (1) *Integrarea strictă a datelor*. Pentru regăsirea și gestiunea datelor, sistemul trebuie să implementeze o interfață uniformă pentru toate instrumentele de acces la date. (2) *Orientarea pe aplicație*. Sistemul trebuie să organizeze datele în structuri capabile să capteze cît mai mult din semantica specifică aplicației. (3) *Integritatea datelor*. Sistemul trebuie să asigure consistența și recuperabilitatea datelor, pentru a garanta astfel satisfacerea constrîngerilor de consistență cerute de aplicație. (4) *Facilizarea accesului*. Sistemul trebuie să asigure o interfață (eventual un limbaj de interogare) pentru a accesa un set de obiecte la un anumit moment. (5) *Independența datelor*. Sistemul trebuie să ascundă structura internă a datelor interfeței de acces, astfel încît dacă structura se modifică, impactul asupra aplicației care utilizează datele să fie cît mai mic.

Detaliile referitoare la distribuire, replicare și concurență trebuie să fie transparente pentru utilizator. Există patru cerințe de transparență care se aplică unui sistem distribuit de obiecte: (1) *Transparența localizării*. Deși datele sunt distribuite pe mai multe noduri și pot migra de pe un nod pe altul, programele nu trebuie să cunoască localizarea lor. (2) *Transparența replicării*. Programele de aplicație trebuie să trateze fiecare entitaite ca un singur element, fără a cunoaște dacă entitatea este sau nu replicată. (3) *Transparența concurenței*. Programele percep propriile lor tranzacții ca fiind singura activitate din sistem. (4) *Transparența defecțiunilor*. Acțiunile tranzacțiilor se materializează fie în totalitate, fie deloc. Dupa realizarea unei tranzacții, efectele ei trebuie să persiste chiar și în cazul unui defect hardware sau a unei erori software.

7.3 Realizarea atomicității tranzacțiilor în sistemele distribuite

S-a aratat în subcapitolul 7.2.1 că conceptul de *tranzacție atomica*, caracterizat prin cele cinci proprietăți cunoscute sub numele de ACID este conceptul fundamental în structurarea sistemelor software tolerante la defecte; importanța să constă în faptul că asigură detectarea erorilor și recuperarea obiectelor. Din punct de vedere al atomicitatii, tranzacțiile se caracterizează în următoarele patru moduri: (1) procesele care execută o acțiune atomică nu au cunostință despre existența altor procese neimplicate în acțiunea atomică și vice versa; (2) procesele care execută o acțiune atomică nu comunică cu alte procese în timp ce se execută acțiunea; (3) procesele care execută o acțiune atomică nu pot detecta o modificare de stare a obiectelor, alta decît cea datorată lor, și nu pot instala o modificare de stare decît cînd acțiunea atomică s-a terminat; (4) o acțiune atomică este indivizibilă și instantanee în sensul că efectul său asupra obiectelor este același, ca și cînd nu ar fi executată concurent cu alte acțiuni.

Dacă în timpul execuției unei tranzacții atomice apare o eroare care împiedică terminarea normală a ei, starea tuturor obiectelor modificate de această tranzacție trebuie restaurată la starea de dinaintea începerii tranzacției; se spune atunci că efectul actualizarilor este anulat și că tranzacția a fost recuperată. În acest sens, o tranzacție ale cărei actualizări pot fi realizate fără a anula efectele altor tranzacții se spune că este *recuperabila*.

Recuperabilitatea și atomicitatea tranzacțiilor pot fi realizate dacă se impun următoarele două cerințe implementării tranzacțiilor: (1) obiectele actualizate nu vor fi eliberate decît la terminarea tranzacției; (2) starea inițială a tuturor obiectelor actualizate de tranzacție poate fi reconstruită. Prima cerința poate fi satisfacută prin utilizarea unui protocol adecvat pentru comiterea tranzacției, iar cea de-a doua poate fi satisfacută prin utilizarea unui mecanism adecvat pentru recuperare.

7.3.1 Protocoale pentru comiterea tranzacțiilor

Figura 7.3.1. arată trei moduri posibile de desfășurare a unei tranzacții. Dacă tranzacția a ajuns în final, se execută o acțiune *commit* pentru a permanentiza modificările asupra obiectelor (7.3.1.a). Dacă tranzacția a detectat condiții pentru care nu poate fi incheiată normal (erori ale datelor de intrare, de exemplu), atunci ea se poate aborta (termină anormal) ea însăși (7.3.1.b). O tranzacție poate fi abortată și de catre sistem datorită unor cauze externe: time-out, rezolvarea unei interblocări, violarea protectiilor, erori pe liniile de comunicație, avarii ale nodului (fig.7.3.1.c.).

Deoarece într-o tranzacție distribuită participa mai multe noduri, este nevoie de un protocol pentru a se obține consensul tuturor participantilor pentru comiterea tranzacțiilor. Se apreciază că problema comiterii tranzacțiilor distribuite este asemănătoare cu problema încercării închiderii unei conexiuni într-un sistem cu linii de

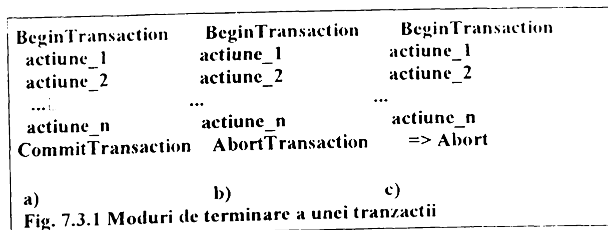


Fig. 7.3.1 Moduri de terminare a unei tranzactii

BUPT

comunicații nesigure; aceasta problemă a fost comparată cu vestita problema a generalilor ([MOO87]). De exemplu, presupunem că o sumă de bani este transferată dintr-un cont în altul din Bucuresti în Timişoara; ambele conturi, contul debitor din Bucuresti şi contul creditor din Timisoara trebuie, fie amîndouă procesate (terminate cu *commit*), fie amîndouă rejectate. Prin renunțarea la condiția ca protocolul să fie finit ca lungime (ceea ce problema generalilor demonstrează că nu este posibil) poate fi găsită o soluție practică la realizarea atomicității tranzacțiilor distribuite.

În acest subcapitol va fi descris *protocolul comiterii tranzacțiilor distribuite în două faze (two phase commit)*. Protocolul presupune existența unui coordonator al comiterii tranzacției, care supervizează comunicația cu toti agenții de la distanță ai unei tranzacții. În cadrul acestui protocol, fiecărui participant trebuie să i se permită să aborteze o tranzacție în orice moment; acest lucru apare din necesitatea ca fiecare participant să-şi poată controla accesul la obiectele dorite. De exemplu, multe mecanisme de control al concurenței (vezi subcapitolul 7.4.) blochează accesul la obiectele accesate de o tranzacție activă. Dacă o tranzacție blochează accesul la un obiect pentru o lung3 perioada de timp, un participant trebuie să poată aborta aceasta tranzacție. Din acest punct de vedere însă, protocoalele de comitere a tranzacțiilor distribuite au o fereastră de vulnerabilitate, în cadrul cărora participantii nu pot aborta unilateral o tranzacție; pentru protocolul comiterii în două faze, aceasta fereastră în cursul căreia nu li se permite participanților să aborteze tranzacție este mică. Protocolul se execută dupa ce toate actualizările asupra obiectelor s-au încheiat, la execuția comenzii CommitTransaction. În acest moment unul dintre participanți devine coordonatorul comiterii tranzacției, în mod obişnuit cel din nodul de la care tranzacția a fost inițiată sau cel dintr-un nod foarte sigur, de exemplu un server central.

Protocolul se desfăşoară, la nivelul coordonatorului, astfel:

Coordonator Faza nr.1:

1. Se trimite un mesaj *PrepareToCommit* tuturor agenților participanți la tranzacție şi se aşteaptă un mesaj de răspuns de la fiecare. Într-un fişier jurnal local, memorat pe un mediu fizic sigur (*stable storage*) se scrie un articol PREPARED.

2.a. Dacă un participant trimite ca mesaj de raspuns *Aborted* sau nu raspunde într-un interval de timp fixat, atunci tranzacția este anulată: coordonatorul trimite un mesaj *AbortTransaction* la toți participantii. Concomitent coordonatorul urmareste desfăşurarea tranzacției înregistrînd în fişierul jurnal local un articol ABORT, asteaptă mesajele de confirmare de la fiecare participant (*AckAbort*) şi tranzacția se termină.

2.b. Dacă toți participanții trimit ca mesaj de raspuns *Prepared*, se trece la faza nr.2.

Coordonator Faza nr.2:

1. Coordonatorul scrie un articol COMMIT în fişierul jurnal şi trimite un mesaj *CommitTransaction* catre toți participanții după care aşteaptă un mesaj de raspuns *AckCommit*, de la fiecare participant.
2. La primirea tuturor mesajelor *AckCommit*, se scrie un articol COMPLET în fişierul jurnal şi tranzacția se termină.

La nivelul participantilor, protocolul se desfăşoară astfel:

Participant Faza nr.1:

1. Se aşteaptă primirea mesajului *PrepareToCommit* de la coordonator.
2. Se blochează obiectele locale implicate în acțiunea atomică, astfel încît să nu mai poată fi accesate de catre alte cereri; totodată starea inițială a datelor se înregistrează pe un mediu fizic sigur (*stable storage*).
3. Dacă pasul 2 a fost realizat cu succes, se scrie un articol PREPARED într-un fişier jurnal local şi se trimite un mesaj *Prepared* coordonatorului; în caz contrar se trimite un mesaj *Aborted* coordonatorului.

Participant Faza nr.2:

1. Se asteapta un mesaj de la coordonator.

2.a. Dacă mesajul primit este *AbortTransaction*, se anulează efectul tranzacției, se eliberează *lock*-urile şi se trimite un mesaj *AckAbort* coordonatorului.

2.b. Dacă mesajul primit este *CommitTransaction*, se eliberează *lock*-urile, se permanentizează actualizările obiectelor, se trimite un mesaj *AckCommit* coordonatorului şi se înregistrează în jurnalul local un articol COMMITED.

În eventualitatea avariei şi apoi a restartării unui nod se realizează următoarele acțiuni: (1) Dacă coordonatorul execută *commit* şi apoi nodul său suferă o avarie, înainte de a scrie articolul COMMIT în fişierul jurnal (pasul 1 al fazei nr.2) atunci trebuie trimis un mesaj AbortTransaction fiecărui participant. (2) Dacă coordonatorul scrie articolul COMMIT dar nodul său suferă o avarie înainte de scrierea unui articol COMPLETE (pasul 2 al fazei nr.2), atunci trebuie trimis un mesaj CommitTransaction fiecărui participant. (3) Dacă coordonatorul suferă o avarie dupa scrierea articolului COMPLETE, atunci la restartare tranzacția se consideră incheiată şi poate fi ignorată.

Protocolul se numeşte în *două faze* deoarece dacă unul dintre participanți nu poate comite acțiunea sa locală sau nodul său suferă o avarie in timpul fazei nr.1, atunci, la restartare, participantul respectiv (mai precis un manager de recuperare din acel nod) va intreba coordonatorul dacă trebuie să anuleze tranzacția neterminată sau să o execute încă o dată.

Deoarece restartările pot avea loc de mai multe ori, operațiile de anulare a operațiilor locale şi de refacere a operațiilor locale (numite *undo* şi respectiv *redo*) trebuie să fie *idempotente*. Dacă coordonatorul suferă o avarie înainte de a scrie in fişierul log, atunci la restartare va transmite *AbortTransaction* tuturor participanților. Nici un participant nu va trece la execuția unui *commit* inainte de primirea unui mesaj *CommitTransaction*. Pe de altă parte, în cazul în care coordonatorul, la restartare, găseşte articolul COMMIT in fişierul jurnal, atunci coordonatorul transmite COMMIT tuturor participanților. Efectul final este că fie toți participanții execută *commit*, fie niciunul.

BUPT

Protocolul *two-phase commit* este adecvat în sistemele în care se dispune de transmisia broadcast a mesajelor şi în care se cere un grad mare de paralelism. Numărul de mesaje pentru n participanti este 4n; acest număr poate fi redus la 2n+2 dacă sistemul dispune de transmisie broadcast sau la 2n dacă mesajele sunt transmise circular, de la un nod la altul; ultimul caz este adecvat pentru sisteme cu următoarele caracteristici: (1) nu este disponibilă transmisia broadcast şi transmisia unui mesaj implică un cost mare; (2) necesităţile de concurenţă sunt reduse; (3) identitatea participanţilor la tranzacţie este de natura statică şi cunoscută.

7.3.2 Protocolul CCR în OSI

Facilităţile de comitere a tranzacţiilor, concurenţă şi recuperare a tranzacţiilor au fost standardizate de ISO în DIS 8649/3 şi DIS 8650/3. Aceste standarde definesc serviciile, respectiv protocolul, pentru componentă OSI, *Commitment, Concurrency and Recovery (CCR)*; serviciile acestei componente CASE sunt disponibile aplicaţiilor fie direct, fie prin intermediul altor componente SASE inrudite (fig.7.3.2.1).

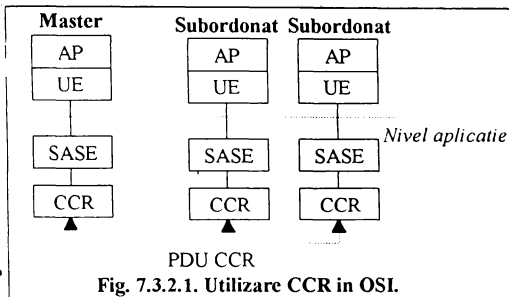


Fig. 7.3.2.1. Utilizare CCR in OSI.

CCR se bazează pe conceptul de tranzacţie atomică şi pe protocolul *two-phase commit* mai sus descris (fig.7.3.2.2). În CCR, iniţiatorul unei tranzacţii este denumit *master* iar agentii participanti *subordonati (slaves)*; înainte de initierea unei tranzacţii trebuie să existe o asociere între aplicaţie (sau SASE) cu rolul de master şi aplicaţiile subordonate (fig.7.3.2.3.a). O tranzacţie atomică este iniţiată de un master printr-o primitiva *C_Begin.request* avînd ca parametru un identificator al tranzacţiei (numele aplicaţiei următ de un sufix unic) şi timpul maxim de aşteptare pînă la transmiterea unei cereri *RollBack* (fig.7.3.2.3). La primirea unei primitive *C_Begin.indication*, fiecare subordonat creează o copie a obiectelor locale şi startează mecanismul de blocare folosit; în practică primitiva *C_Begin* are efectul stabilirii la nivelul sesiunii a unor puncte de sincronizare pentru aceasta asociere. În continuare, se transmit şi se execută operaţiile specifice aplicaţiei, utilizînd primitive SASE. În fig.7.3.2.3 se arată primitivele CCR pentru cele două cazuri posibile de terminare a unei tranzacţii: normal (b) sau anormal (c). În primul caz, (b), toţi subordonatii returnează succes la cererea de *commit*, utilizînd serviciul *C_Ready*; în cel de-al doilea caz, (c), unul sau mai mulţi subordonati returnează un raspuns negativ masterului, utilizînd serviciul *C_Refuse*. În practica, serviciul *C_Prepare* este opţional şi, dacă nu este utilizat, subordonaţii indică disponibilitatea lor printr-un *C_Ready* sau *C_Refuse* imediat dupa terminarea operaţiilor aplicaţiilor. În plus, există şi serviciul *C_Restart* care poate fi folosit atît de un master cît şi de un subordonat, pentru a aduce obiectele implicate într-o tranzacţie la o stare anterioară; poate fi folosit însă de un subordonat numai dacă acesta nu şi-a dat acordul pentru *commit*. Un master determină starea curentă a unui subordonat executînd primitiva *C_Restart.request* cu un parametru care indică punctul de reluare astfel: (1) ACTION, semnifică restartarea acţiunii atomice la început; (2) COMMIT, semnifică restartarea acţiunii atomice dupa ultimul punct COMMIT; (3) ROLLBACK, semnifică restartarea acţiunii atomice dupa ultimul punct ROLLBACK. Similar, punctul de reluare dintr-o primitivă de confirmare se indică astfel: (1) DONE, semnifică faptul că subordonatul a realizat ultima cerere de *rollback* sau *commit*; (2) REFUSE, semnifică faptul că subordonatul nu a realizat ultima cerere şi de aceea masterul trebuie să execute o cerere *rollback*; (3) ACTION, semnifică faptul că subordonatul trebuie să restarteze acţiunea de la început; (4) RETRY_LATER, semnifică faptul că subordonatul nu poate continua cu o restartare în acest moment şi că masterul trebuie să încerce mai tarziu.

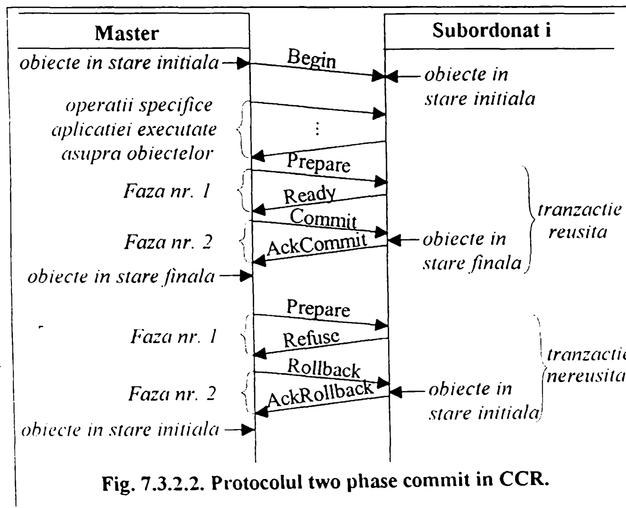


Fig. 7.3.2.2. Protocolul two phase commit in CCR.

Protocolul CCR poate genera un arbore de tranzacţii distribuite, în care un subordonat se comportă ca un master pentru o altă tranzacţie atomică descrisă prin subarborele respectiv. Un parametru, identificatorul de ramură, se utilizează pentru a specifica o ramură din aceasta ierarhie. Dacă unul sau mai mulţi subordonati refuză o propunere *commit*, refuzul se propagă pînă la masterul arborelui; acesta execută atunci o cerere *rollback*, care se propagă în jos la toţi subordonaţii (fig.7.3.2.4).

BUPT

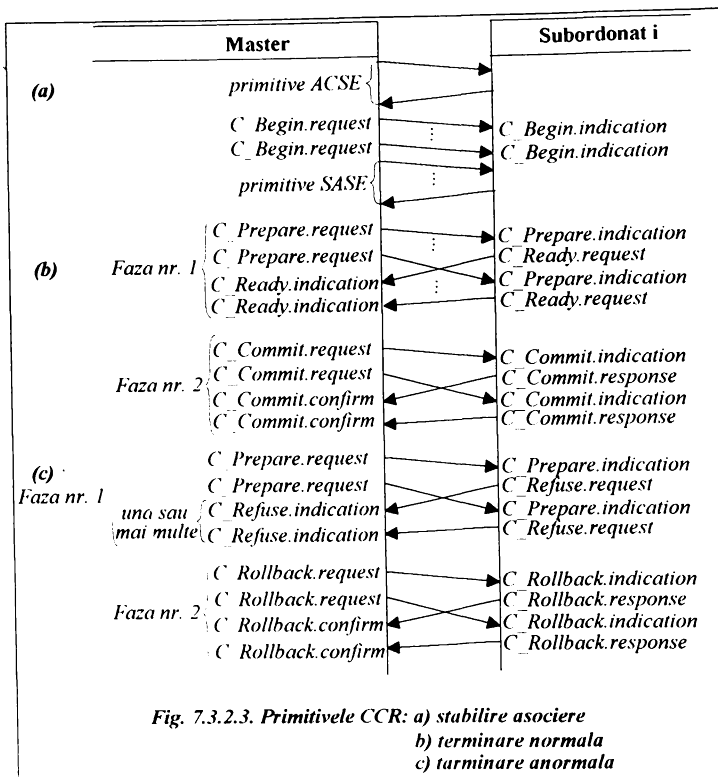


Fig. 7.3.2.3. Primitivele CCR: a) stabilire asociere
b) terminare normala
c) tarminare anormala

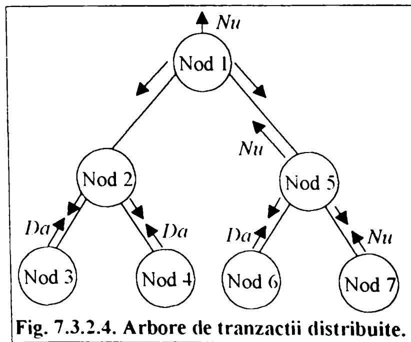


Fig. 7.3.2.4. Arbore de tranzactii distribuite.

7.3.3. Implementarea tranzacţiilor atomice

Cînd o tranzacţie este in desfăşurare, actualizările sale asupra obiectelor trebuie făcute într-o manieră reversibilă. Rezultă că participanţii trebuie să menţina două stări ale obiectelor implicate într-o tranzacţie; cea *iniţială*, premergatoare tranzacţiei, care trebuie restaurată în cazul unui incident hardware sau a unei abortări a tranzacţiei şi cea *intermediară*, inconsistentă, modificată pe masură ce tranzacţia evoluează. Cînd un participant recepţionează din partea coordonatorului cererea de comitere a tranzacţiei, starea intermediară este considerată stabilă şi înlocuieşte starea iniţială. Pe de altă parte, mecanismele de control al concurenţei pot impune uneori lucrul cu o copie a obiectelor şi nu cu obiectele propriu-zise (vezi subcapitolul 7.4.6.). Există două implementări mai cunoscute ale tranzacţiilor atomice care respectă aceste cerinţe: prima implementare este cunoscută sub numele de *tehnica paginării umbra*; cea de-a două implementare este cunoscută sub numele de *tehnica jurnalelor undo/redo*.

În cadrul primei tehnici, tranzacţia nu operează direct asupra obiectelor, ci asupra unor copii ale obiectelor situate într-o zonă de lucru a tranzacţiei. Pînă în momentul comiterii tranzacţiei toate operaţiile asupra obiectelor se execută asupra copiilor obiectelor din zona de lucru şi nu asupra obiectelor propriu-zise. Dacă, de exemplu, obiectele

BUPT

sunt fişiere, atunci, înainte de a actualiza un bloc al unui fişier, se obţine un bloc liber din lista de blocuri libere, în care se copiază vechiul bloc al fişierului şi se actualizează aceasta copie; concomitent, se copiază în zona de lucru şi se actualizează şi harta blocurilor alocate fişierului. Astfel, în timp ce tranzacţia respectivă lucrează cu o noua hartă de alocare a fişierelor, celelalte aplicaţii percep fişierul ca neschimbat. Noile blocuri alocate fişierului se numesc *pagini umbra*. Fig.7.3.3.1 ilustrează această tehnică. Se observă că se utilizează şi un fişier jurnal, *jurnal de intenţii*, care înregistrează toate blocurile umbră introduse de o tranzacţie. La comiterea tranzacţiei, harta fişierului se actualizează, iar spaţiul ocupat de vechile blocuri ale fişierului se eliberează. Dacă are loc un incident hardware în timpul comiterii tranzacţiei, atunci, la repunerea în funcţiune, avînd înregistrat jurnalul de intenţii (pe un mediu fizic sigur-*stable storage*) orice participant va actualiza hartile de fişier

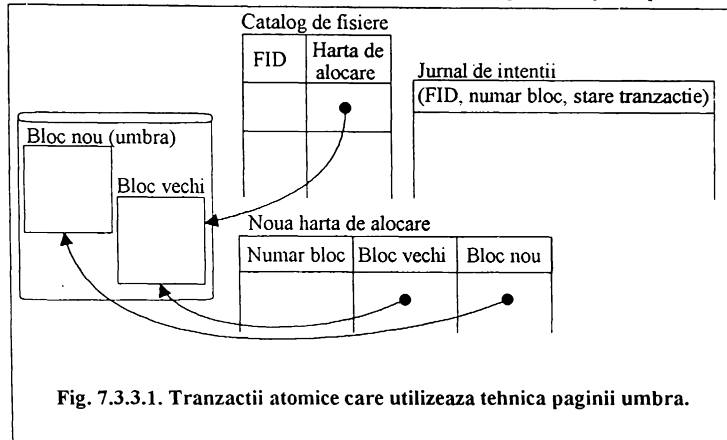


Fig. 7.3.3.1. Tranzactii atomice care utilizeaza tehnica paginii umbra.

conform jurnalului de intenţii.

O altă tehnică de implementare a tranzacţiilor atomice este *tehnica jurnalelor undo/redo* numită uneori *tehnica listelor de intenţii*. Aceasta elimină redundanţa tehnicii paginii-umbra plecînd de la două premise: (1) în general, majoritatea tranzacţiilor nu esueaza; (2) regia de sistem sporită în cazul tehnicii paginii-umbră se datorează copiilor de blocuri de disc cu care se lucrează, ocolind blocurile alocate. Tehnica jurnalelor undo/redo permite efectuarea actualizarilor direct asupra obiectelor; dar, înainte de modificarea unui obiect, valoarea veche şi noua a acestuia se memorează într-un fişier jurnal; o intrare în acest fişier va conţine deci identificatorul tranzacţiei, identificatorul obiectului, vechea şi noua valoare. Articolele fişierului jurnal pot fi repertorizate într-un fişier hartă a jurnalului (fig.7.3.3.2).

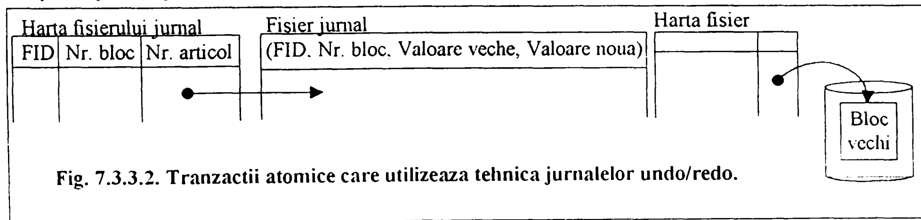


Fig. 7.3.3.2. Tranzactii atomice care utilizeaza tehnica jurnalelor undo/redo.

În afară de regia de sistem mai redusa, tehnica jurnalelor undo/redo mai are şi următoarele avantaje faţă de tehnica paginii-umbră: (1) menţine alocarea iniţială a blocurilor unui fişier, pastrînd eventual contiguitatea blocurilor, chiar şi în cazul unei actualizări într-o ordine aleatoare a blocurilor, ceea ce nu este adevarat în tehnica paginii-umbră; (2) poate fi construită ca un nivel deasupra sistemului de gestiune al fişierelor, deoarece nu implică acces la blocuri şi la hartile de alocare ale fişierelor.

7.3.4. Recuperarea obiectelor

Recuperarea obiectelor se defineşte ca fiind tehnica prin care se restaurează o stare consistentă anterioară a unui sistem de obiecte, în cazul unei defecţiuni hardware sau a unei erori software.

Soluţia tradiţională în bazele de date centralizate este crearea periodică a unor puncte de reluare şi recuperarea prin revenirea la cel mai recent punct de reluare. Într-un sistem distribuit însă această soluţie se complică datorită cerinţei de respectare a consistenţei datelor în sistemul distribuit şi necesităţii de recuperare în urma oricăror combinaţii de erori ale liniilor de comunicaţii, ale nodurilor, erori software, incluzînd chiar erorile din timpul realizării recuperării.

Se lucrează de aceea cu un *sistem de recuperare al obiectului*, bazat pe un *fişier jurnal (log)*; acesta conţine o istorie a tuturor acţiunilor recuperabile (al căror efect se poate anula) şi implementează *abortarea unei tranzacţii*. Recuperarea obiectelor este strîns legată de modul de implementare al tranzacţiilor prezentat în subcapitolul anterior.

BUPT

7.3.4.1. Protocolul Write-Ahead Logging

Un fișier *log* este un fișier secvential, potențial nelimitat ca lungime, care poate fi modificat numai prin adăugarea de articole. În practică, articolele care trebuie scrise în log sunt în mod obisnuit bufferate în memoria principală, în timp ce pe disc se rezervă o dimensiune fixă. Cînd dimensiunea devine prea mare pentru ca articolele să poată începe în aceasta zonă, fie se salvează o parte necritică a conținutului într-o zonă offline, fie se trunchează periodic fișierul astfel încît să nu se piardă informație critică. Scrierea într-un fișier log se face într-o manieră atomică, utilizînd tehnica *stable storage*. În mod obișnuit, tehnica *stable storage*, fie că utilizează instructiuni de I/O la nivel fizic (operațiile de bază furnizate de hardware), fie că utilizează apeluri ale sistemului de operare pentru scriere (dar în sistemul de operare UNIX, de exemplu, acestea trebuie urmate și de apeluri fsync() pentru a asigura scrierea efectivă pe disc) realizează două copii ale datelor. Ca atare, dacă în timpul unei operații de scriere apare un incident hardware, cel puțin una dintre copii rămîne validă, și astfel și cealaltă poate fi corectată (sau creată din nou).

În general, scrierea articolelor în fișierele log se face urmarindu-se două scopuri:(1) pentru a anula efectul unor tranzacții abortate;(2) pentru a restabili efectul unor tranzacții care au executat *commit*, dar ale căror actualizări au fost întrerupte sau distruse de un incident hardware.

Protocolul Write-Ahead Logging (WAL) este un protocol care definește momentele în care articolele log, bufferate în memorie, trebuie scrise pe disc, cu scopul de a asigura atomicitatea tranzacțiilor. Informațiile pentru anularea tranzacțiilor (*undo* information) trebuie scrise pe disc înainte de modificarea versiunii persistente a unui obiect printr-o operație a unei tranzacții. Informațiile pentru restabilirea efectului unei tranzacții (*redo* information) trebuie scrise pe disc înainte ca tranzacția să execute *commit*. Acest protocol oferă avantaje unanim recunoscute față de alte tehnici pentru recuperare. Cînd o tranzacție accesează date ale unui obiect (definit de utilizator), se crează în memorie o copie a acelui obiect; dacă obiectul este modificat, copia respectivă se rescrie pe disc (versiunea persistentă a obiectului). Deoarece informațiile necesare pentru anularea actualizarilor tranzacțiilor sunt scrise în log, versiunea persistentă a obiectului poate fi *oricînd* înlocuită pe disc cu o copie actualizată a obiectului, chiar dacă aceasta reflectă actualizări ale unei tranzacții încă necomise. Pentru alte mecanisme (ca de exemplu tehnica paginii umbră, prezentată în subcapitolul precedent) alocatorul de memorie trebuie să rezerve un nou spațiu pe disc pentru noua versiune actualizată a obiectului; versiunea nemodificată a obiectului trebuie reținută pentru ca tranzacția să poată fi abortată. Astfel, deoarece obiectele actualizate își mută poziția pe disc, strategiile pentru plasarea grupată a obiectelor înrudite (*clustering strategies*) sunt dificil de implementat (dezavantajul (1) al tehnicii paginii-umbră din subcapitolul precedent). Analog, deoarece fișierul log conține și informații pentru restabilirea efectului unei tranzacții, obiectele actualizate nu trebuie obligatoriu scrise pe disc înainte de comiterea unei tranzacții. Obiectele frecvent actualizate pot rămîne în memoria principală atîta timp cît este necesar, fără a fi rescrise pe disc (obiectul din memorie joacă astfel rolul de cache).

În plus, în afară de articolele utilizate pentru informațiile *redo* și *undo*, în fișierele log se mai înregistrează și alte tipuri de informații: articole *checkpoint*, care conțin informații pentru a limita căutarea în fișierul log în cursul unei recuperari, articole care descriu modificarea stării tranzacțiilor în decursul unui protocol *two-phase commit* și, eventual, informații de stare necesare mecanismului de control al concurenței care trebuie reținute și dupa incidentele hardware.

7.3.4.2. Tehnicile Value-Logging și Operation Logging

Tehnica Value-Logging este cea mai simplă tehnică pentru recuperarea obiectelor, utilizînd protocolul WAL: cînd se actualizează un obiect, sistemul de recuperare scrie în fișierul log *valoarea* obiectului neactualizat și *valoarea* obiectului actualizat; trebuie facută observația că se poate scrie în fișierul log și numai un cîmp al valorii obiectului (cîmpul care se actualizează), și nu toate cîmpurile, pentru economie de spațiu. Dacă tranzacția va executa *commit*, și apare un incident hardware, sistemul de recuperare trebuie să restaureze valoarea obiectului la cea de dinaintea operațiilor de actualizare, pe care o gasește în fișierul log. Dacă tranzacția execută *commit*, și apare un incident hardware care împiedică actualizarea versiunii persistente a obiectului, sistemul de recuperare trebuie să restaureze valoarea obiectului la cea finală de dupa toate operațiile de actualizare, pe care o gaseste în fișierul log. Este evident că operațiile facute de sistemul de recuperare a obiectelor, în cazul acestei tehnici, pentru a anula și reinstală efectul unei tranzacții sunt *idempotente*: execuția lor de mai multe ori, în cazul unor incidente în cursul procesului de recuperare are același efect.

Mecanismul de control al concurenței, care coordonează accesul la obiecte a mai multor tranzacții, trebuie să se asigure că dacă un obiect este actualizat de o tranzacție, el nu va mai fi prelucrat de nici o altă tranzacție pînă la comiterea primei tranzacții. În caz contrar, dacă prima tranzacție ar fi abortată după cel puțin o actualizare și deci efectul ei anulat, atunci ar trebui abortate toate tranzacțiile care au accesat acel obiect, chiar și cele care între timp au fost comise. Figura 7.3.4.1. ilustrează această situație. Înainte de incidentul hardware, T_3 actualizează obiectul O_1 și deblochează accesul la acest obiect; în continuare T_4 citește obiectul O_1 și actualizează obiectul O_2, apoi se termină normal cu *commit*. După repararea incidentului, T_3 poate fi abortată și apoi restartată, dar T_4 este deja terminată și nu mai poate fi abortată, chiar dacă a realizat, în mod eronat, actualizarea obiectului O_2 în funcție de valoare lui O_1. Acest efect se numește *efect de domino* sau *abort în cascada al tranzacțiilor*.

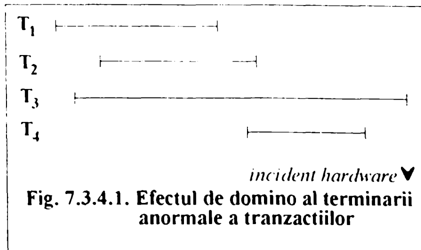


Fig. 7.3.4.1. Efectul de domino al terminarii anormale a tranzactiilor

BUPT

În multe situații, totuși, tehnica value-logging nu poate fi folosită datorită limitării severe a performanțelor. De exemplu, dacă se înregistrează mai multe articole ale unui fișier într-o pagină și se consideră o pagină ca un obiect, atunci cînd se actualizează un obiect din acea pagină de catre o tranzacție, nici o altă tranzacție nu va mai putea utiliza nu numai acel articol, ci orice alt articol din acea pagină, pînă la comiterea primei tranzacții; aceasta restricționează într-un mod acceptabil concurența.

Tehnica Operation Logging poate fi utilizată pentru a diminua restricțiile cu privire la actualizările efectuate de tranzacții neterminate. În cadrul acestei tehnici, sistemul de recuperare scrie în fișierul log, *o descriere a operației* care a provocat actualizarea unui obiect și nu valoarea obiectului de dinainte și de după actualizare. Atîta timp cît pentru fiecare operație există și o operație compensatorie, care să anuleze efectul operației, o tranzacție poate fi întotdeauna abortată. Operația poate fi și reexecutată, cînd un incident hardware apare după *commit*. Datorită faptului că reprezentarea unei operații în fișierul log se face mult mai compact decît reprezentarea valorii obiectelor, tehnica Operation logging prezintă avantajul economiei de spațiu în log și al unui timp mai mic de scriere în log.

Se dau următoarele exemple de operații care sunt adecvate pentru aceasta tehnică: operații de incrementare și decrementare a unui contor, de inserare și ștergere a unui nod într-un arbore B care poate memora, de exemplu, maparea numelor de fișiere din cataloage în identificatori de fișiere sau relațiile de indexare dintr-o bază de date.

Deși tehnica operation logging oferă anumite avantaje, totuși introduce un nou gen de pronleme. Multe din operațiile care actualizează obiectele *nu sunt idempotente*; dacă în cursul unui proces de recuperare se repetă aceste operații, rezultatele obținute vor fi eronate. Cea mai cunoscută soluție la acest tip de problemă este de a asocia fiecărui obiect *un număr de secvență log*, cu valori strict crescatoare, care va reflecta starea obiectului scris pe disc. Înainte de a repeta o actualizare, se compară acest număr din log cu numărul asociat obiectului; dacă numărul asociat obiectului este mai mare sau egal cu cel din log, actualizarea descrisă în log a fost deja aplicată și nu mai trebuie reinstalată.

7.3.4.3. Tehnicile Redo-Only Logging și Undo-Only Logging

În unele situații poate fi avantajos să se întîrzie actualizările unei tranzacții pînă în momentul comiterii ei; acest lucru asigură faptul că operațiile tranzacției nu vor trebui să fie niciodată anulate și de aceea vechile valori ale obiectelor nu se salvează în log. Aceasta tehnică, numită *tehnica redo-only logging* este avantajoasă, de exemplu, în cazul unei operații de ștergere a conținutului unui fișier sau a unui articol al unui fișier. Operațiile de ștergere se reexecută simplu, dar efectul lor se anulează greu; informațiile de *undo* pentru aceste operații sunt de dimensiuni mari. Dacă, în schimb, actualizările sunt întîrziate pînă în momentul comiterii tranzacției, astfel încît informațiile de *undo* nu trebuie scrise în log, sistemul trebuie să garanteze că, odată ce tranzacția execută *commit*, orice actualizări întîrziate sunt realizate, chiar dacă intervine un incident hardware. Tehnica generală este de a crea o *lista de intentii* care să cuprindă actualizările în așteptare și de a forța scrierea acestor liste în log, înainte de faza *prepare to commit*. Există sisteme de fișiere a căror tehnică de recuperare se bazează în întregime pe tehnica redo-only logging: sistemele de fișiere Alpine ([BKT85]) și Oregon ([Hag87]) interzic rescrierea obiectelor pe disc pînă în momentul în care tranzacția execută *commit*.

O altă tehnică utilă este de a scrie în fișierul log numai informațiile *undo* și de a forța scrierea pe disc a obiectelor actualizate înainte de *commit*. Aceasta tehnică, numită *tehnica undo-only logging* asigură faptul că actualizările nu vor trebui niciodată reexecutate și, de aceea, noile valori ale obiectelor nu se scriu în log. Tehnica este utilă pentru cazul în care se crează fișiere noi, într-un sistem de fișiere.

Trebuie specificat, în final, că, în general, un sistem de recuperare poate folosi o combinație de tehnici de recuperare prezentate în subcapitolele 7.3.4.2. și 7.3.4.3., deci o tehnică hibrida. În acest sens exemple sugestive sunt sistemul de fișiere QuickSilver Distributed Fie System (QDFS) și sistemul de gestiune de baze de date Starbust ([CPS93]). Astfel, aceste sisteme folosesc atît tehnica *value-logging* (in Starbust, pentru înregistrarea continutului bazei de date, în QDFS pentru înregistrarea hărții de alocare a fișierelor), cît și tehnicile *operation-logging* (pentru inserarea sau ștergerea cheilor într-un arbore B), *redo-only logging* (pentru ștergerea fișierelor în DFS și a relatiilor și fișierelor de relatii în Starbust) și *undo-only logging* (pentru crearea fișierelor noi care conțin articole ale bazei de date).

7.3.4.4. Structura sistemului de recuperare

Un sistem de recuperare al obiectelor în cadrul modelului tranzacțional prezentat în subcapitolul 7.2.2. are structura generală prezentată în fig. 7.3.4.4.

El este compus din două elemente principale:

1. Un *manager de recuperare* al cărui rol este de a urmări evolutia tranzacțiilor, și de a lua deciziile adecvate conform tehnicii de recuperare alese, în timpul execuției acțiunilor tranzacției, ca și la execuția comenzilor *commit, abort*.
2. Un *manager pentru log*, utilizat de manageul de recuperare și de alte componente ale sistemului distribuit pentru a scrie articole în log; scrierea articolelor în fișierul log este cerută fie de managerul de recuperare, fie de sistem.

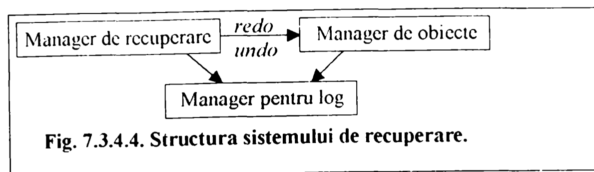


Fig. 7.3.4.4. Structura sistemului de recuperare.

BUPT

7.4. Controlul concurenţei

În scopul menţinerii consistenţei datelor distribuite, se impun anumite constrîngeri în structura tranzacţiilor sau în maniera de realizare a concurenţei.

7.4.1. Cerinţele mecanismului de control al concurenţei

Kohler ([Koh81]) a arătat că, dacă într-un sistem distribuit nu apare o defecţiune hardware, *toate mecanismele de control al concurenţei trebuie să asigure următoarele:*

(1) consistenţa obiectelor;
(2) realizarea fiecărei tranzacţii atomice într-un timp finit.

În plus, un *bun* mecanism de control al concurenţei se caracterizează astfel:

(1) permite execuţia paralela a acţiunilor, în scopul satisfacerii cerinţelor de performanta ale sistemelor;
(2) implică prelucrări minore;
(3) se execută cu performante satisfacatoare chiar şi în reţele în care întîrzierile de transmisie sunt semnificative;
(4) impune puţine constrîngeri în structura tranzacţiilor.

7.4.2 Serializarea tranzacţiilor

În continuarea acestui capitol se va folosi următoarea definiţie formală pentru o tranzacţie.

Definiţie:

O *tranzacţie* T_i este o pereche $(T_i, <_{T_i})$, în care $T_i=\{T_{ij}: 1<=j<=m_i\}$ este setul de acţiuni aparţinînd tranzacţiei i iar $<_{T_i}$ este o relaţie de precedenţă care ordonează setul T_i.

Pentru a se preciza acţiunea ce se execută asupra sistemului de obiecte şi obiectul asupra căruia se execută (in modelul prezentat la 7.2.2) se vor utiliza următoarele notaţii:

citire: $<T_i$.read_object$_j(X,S_K)$,V>
scriere: $<T_i$.write_object$_j(X,S_K)$,V>

în care prin T_i se precizează tranzacţia, prin *read_obiect$_j$*, respectiv *write_obiect$_j$* se precizează acţiunea, prin X obiectul, prin S_K nodul pe care se afla obiectul iar prin V valoarea rezultată prin citire, respectiv implicată în scriere.

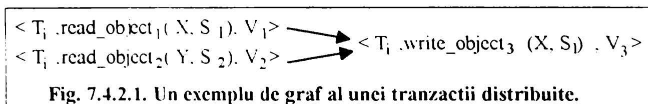


Fig. 7.4.2.1. Un exemplu de graf al unei tranzactii distribuite.

O tranzacţie poate fi reprezentată printr-un graf G=(N,A) in care N este un set de noduri corespunzînd acţiunilor tranzacţiei T_i iar $A=\{(T_{ij},T_{iK}):T_{ij}<_{T_i}T_{iK}\}$ este un set de arce indicînd ordinea de execuţie a acţiunilor. În figura 7.4.2.1 este prezentat un exemplu de graf al unei tranzacţii simple. Acest graf reprezintă tranzacţia T_i compusă din acţiuni asupra a trei entităţi (sau obiecte, deoarece entităţile nu sunt replicate). Acţiunea *write* trebuie executată dupa terminarea ambelor acţiuni de citire; nu există restricţii asupra ordinii de execuţie a celor două citiri.

Pot fi utilizate două modele pentru tranzacţiile distribuite: *modelul tranzacţiilor seriale* şi *modelul tranzacţiilor concurente*. În modelul tranzacţiei seriale se presupune că relatia de precedenţă $<_{T_i}$ ordonează total seturile T_i; în modelul tranzacţiei concurente, se presupune că relatia $<_{T_i}$ ordonează parţial seturile T_i. În figurile 7.4.2.2 şi 7.4.2.3 se prezinta două exemple de grafuri pentru cele două modele.

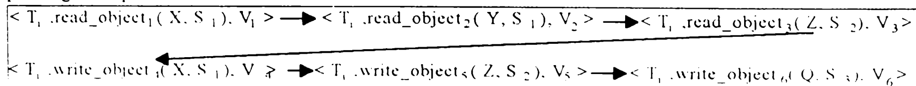


Fig. 7.4.2.2. Un exemplu de graf al unei tranzactii distribuite seriale.

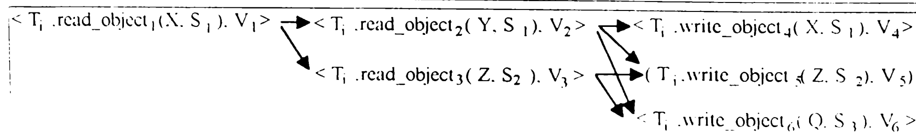


Fig. 7.4.2.3. Un exemplu de graf al unei tranzactii distribuite concurente.

S-a arătat în subcapitolul 7.2 că principala cauză pentru situaţiile de violare a consistentei sistemelor la obiecte este execuţia concurentă a unui set de acţiuni provenind de la tranzacţii diferite. Controlul concurenţei trebuie să actioneze prin intermixarea corectă a acestor acţiuni; controlul concurenţei va consta în planificarea acţiunilor tranzacţiilor. Altfel spus, orice planificare a acţiunilor unui set de tranzacţii este rezultatul aplicării *algoritmului de control al concurenţei* setului de tranzacţii.

BUPT

Definiție:

O *planificare a unui set de tranzacții* τ este o mulțime parțial ordonată $(T(\tau),<_S)$ în care $T(\tau)=U_iT_iUT_0UT_f$ este mulțimea tuturor acțiunilor executate de tranzacții, completată cu încă două tranzacții, o *tranzacție inițială* T_0 care scrie (stabileste) starea inițiala a sistemului distribuit de obiecte și o *tranzacție finală* T_f care citește starea finală iar $<_S$ este o relație de ordine parțială peste $T(\tau)$ pentru care sunt îndeplinite următoarele condiții:

(1) $<_S \; U_i<_{Ti}$

(2) pentru orice pereche de acțiuni T_{ij} din $T(\tau)$, T_{Kl} din $T(\tau)$, cu i<>K, care cer acces la același obiect, dintre care cel puțin una este de scriere, este adevarată relația:

$T_{ij}<_S T_{Kl}$ sau $T_{ij}<_S T_{Kl}$

În contrast cu sistemele centralizate, într-un sistem distribuit este imposibil să se determine la un moment dat starea globală a întregului sistem sau să se determine aprioric ordinea de execuție a acțiunilor în prezența întîrzierilor de transmisie. De aceea, relatia $<_S$, care desemnează ordinea de execuție a acțiunilor într-o planificare $S(\tau)$ este o relație de ordine partiala. Condiția (1) impune ca relatia $<_S$ să respecte ordonarea acțiunilor stipulată în cadrul tranzacțiilor. Condiția (2) impune ca acțiunile *read* și *write* sau *write* și *write* asupra aceluiași element al unui obiect să fie executate serial.

O planificare $S(\tau)$ poate fi reprezentată printr-un *graf al planificarii.*

Definiție:

Un *graf al unei planificări $SG(S(\tau))$* este un graf orientat (N,A) ale cărui noduri corespund tuturor acțiunilor $T(\tau)$ și ale cărui arce sunt toate perechile (T_{ij},T_{Kl}) astfel încît $T_{ij}<_S T_{Kl}$.

Planificările pot fi *seriale* sau *concurente.* O planificare $S(\tau)$ este *serială* dacă pentru orice două tranzacții, toate acțiunile unei tranzacții se execută înaintea tuturor acțiunilor celei de-a doua tranzacții; în caz contrar, cînd acțiunile unor tranzacții diferite se execută în paralel (sau intercalat), planificarea este *concurentă.* Cu alte cuvinte, o planificare conținînd tranzacțiile T_1, T_2,...T_m este serială (și consistentă) dacă pentru orice i=1,... m-1 tranzacția T_i se execută complet înainte ca tranzacția T_{i+1} să-și înceapă execuția.

O planificare a unui set de tranzacții τ executate în N noduri într-un sistem distribuit poate fi reprezentată printr-un set de planificări locale $S(\tau)=\{S_1, S_2,... S_N\}$ care descriu execuția tranzacțiilor la sistemele locale de obiecte.

Formal, o planificare locală S_k a unui set de tranzacții τ în nodul S_k este o mulțime parțial ordonată $(T^{sk}(\tau),<)$ în care $T^{sk}(\tau)=\{T_{ij}: T_{ij}$ se execută în nodul $S_k\} \in T(\tau)$. Se consideră în continuare problema corectitudinii planificarilor tranzacțiilor în contextul unor *informatii sintactice* asupra tranzacțiilor; aceasta înseamnă că algoritmul de control al concurenței are la dispoziție numai informații despre identitatea elementelor obiectelor la care se cere acces (modelul *controlului sintactic al concurenței*). Un criteriu firesc de corectitudine pentru planificările concurente este *criteriul de serializare al tranzacțiilor.* Înainte de a enunța acest criteriu se mai introduc următoarele definiții:

Definiții:

(1) Se numește *stare* a unui sistem de obiecte mulțimea de valori asignate elementelor tuturor obiectelor din sistemul distribuit de obiecte.

(2) Se numește *imaginea* sistemului de obiecte percepută de o tranzacție T_i într-o planificare $S(\tau)$ setul de valori ale datelor citite de acțiunile tranzacției.

(3) Două planificări $S(\tau)$ = $(T(\tau), <_s)$ și $S'(\tau)=(T(\tau),<_s)$ sunt *echivalente din punct de vedere al imaginilor* dacă imaginea percepută de orice tranzacție $T_i \in S(\tau)$ este identică cu imaginea percepută de aceeași tranzacție în planificarea $S'(\tau)$.

(4) Două planificări $S(\tau)=(T(\tau), <_s)$ și $S'(\tau)=(T(\tau),<_s)$ sunt *echivalente din punct de vedere al starilor,* dacă pentru orice prelucrări ale tranzacțiilor conținute în τ, stările finale obținute de S și S' sunt identice.

(5) Două planificări $S(\tau)$ = $(T(\tau), <_s)$ și $S'(\tau)$ = $(T(\tau),<_s)$ sunt *echivalente* dacă ele sunt echivalente ca imagine și ca stări.

Prin urmare, două planificări S1 și S2 sunt echivalente dacă: (1) setul de tranzacții din S_1 și S_2 este identic; (2) pentru orice obiect X din S_1, dacă tranzacția T_i execută o citire a obiectului X și valoarea lui X citită de T_i este modificată de T_j, aceeași situație trebuie să apară și în S_2 (sincronizare *read-write*); (3) Pentru fiecare obiect X din S_1, dacă tranzacția T_i execută o scriere a obiectului X, înainte ca T_j să execute și ea o scriere a lui X, aceeași situație trebuie să apară și in S_2 (sincronizare *write-write*).

Criteriu de serializare

O planificare $S(\tau)$ este *corecta* dacă este echivalentă cu o planificare serială a setului de tranzacții .

O planificare echivalentă cu o planificare serială se numește *planificare serializabila*; mulțimea tuturor planificarilor serializabile ale unui set de tranzacții τ se va nota cu SR. În figura 7.4.2.5. se arată un graf al

BUPT

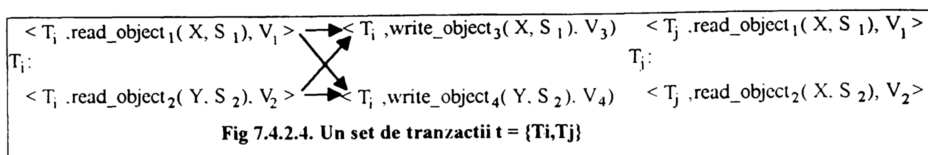


Fig 7.4.2.4. Un set de tranzactii t = {Ti,Tj}

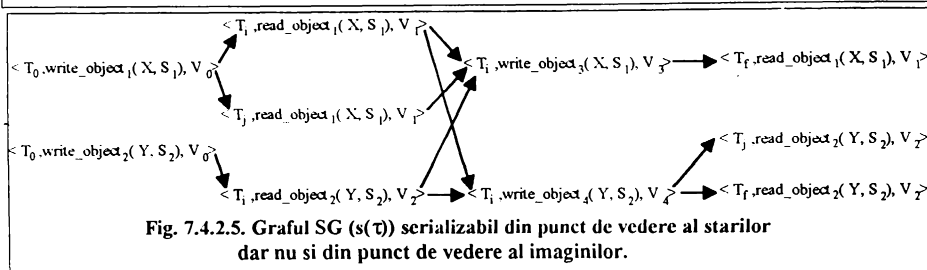


Fig. 7.4.2.5. Graful SG (s(τ)) serializabil din punct de vedere al starilor dar nu si din punct de vedere al imaginilor.

tranzacțiilor din figura 7.4.2.4. serializabil din punct de vedere al starilor, dar neserializabil din punct de vedere al imaginilor deoarece tranzacția T_j citește valoarea obiectului X, care este inconsistentă.

Pentru a verifica serializabilitatea unei planificări concurente, S(τ), trebuie examinată ordinea acceselor la date al acțiunilor tranzacțiilor. Verificarea se face utilizînd un *graf de serializare*.

Definiție:

Graful de serializare al planificării S(τ), notat SRG(S(τ))=(N,A) este un graf orientat, a cărui mulțime de noduri N corespunde setului de tranzacții extins cu două tranzacții ipotetice T_o și T_f, care stabilesc starea inițială, respectiv starea finală și a cărui mulțime de arce se definește astfel: dacă există un obiect X și o acțiune write_object$_j$(X)T_i și acțiune read_object$_l$(X) T_k astfel încît a doua acțiune citește valoarea scrisă de prima, atunci:

(1) $(T_i,T_k)\in A$

(2) dacă $T_i\neq T_o$, $T_k\neq T_f$ și există o tranzacție $T_p\neq T_o$ care scrie pe X, atunci fie $(T_p,T_i)\in A$ fie $(T_k,T_p)\in A$.

(3) dacă $T_i\neq T_o$, atunci $(T_o,T_i)\in A$.

(4) dacă $T_i=T_o$, $T_k\neq T_f$ și există o tranzacție $T_p\neq T_o$ care scrie pe X, atunci $(T_k,T_p)\in A$.

(5) dacă $T_k=T_f$ și există o tranzacție T_p care scrie pe X atunci $(T_p,T_i)\in A$.

Prin condiția (1) se impune ca, dacă într-o planificare S() o tranzacție T_k citește valoarea lui X scrisă de o tranzacție T_i, atunci T_i trebuie să preceadă T_k în orice planificare serială echivalentă; precedența este reprezentată prin arcul (T_i,T_k). Condiția (2) impune ca, dacă într-o planificare S(τ), o tranzacție T_k citește o valoare scrisă de T_i și o altă tranzacție T_p scrie X, atunci T_p trebuie fie să preceadă pe T_i, fie să urmeze lui T_k în orice planificare serială echivalentă. Condițiile (3),(4),(5) se referă la tranzacția inițială și la cea finală. Dacă T_i este tranzacția inițială, atunci nici o tranzacție T_p nu o poate precede; de asemenea, dacă T_k este tranzacția finală, nici o tranzacție T_p nu poate să o urmeze.

Teorema:

O planificare S(τ) este *serializabila* dacă și numai dacă poate fi construit un graf de serializare SRG(S(τ)) aciclic.

Deoarece verificarea existenței unui astfel de graf implică construcția tuturor grafurilor de serializare, o problemă NP completă în practică, soluțiile propuse pentru controlul concurenței utilizează un alt criteriu de corectitudine, mai restrictiv, cel al *serializabilității D*. Pentru a defini serializabilitatea D, trebuie definite în prealabil următoarele elemente:

Definiții:

(1) Două acțiuni T_{ij} și T_{kl} sunt în conflict în accesul la obiectul X, conflict desemnat de T_{ij}-->$^C{}_X T_{kl}$, dacă ambele accesează același obiect și cel puțin o acțiune este de scriere.

(2) Două tranzacții sunt în conflict de acces la date, conflict specificat prin T_i-->$^C T_j$ dacă există un obiect X și acțiunile $T_{ij}\in T_i$ și $T_{kl}\in T_k$ astfel încît T_{ij}-->$^C{}_X T_{kl}$.

(3) Acțiunea T_{ij} precedă acțiunea T_{kl} în contextul planificării S(τ) în raport cu obiectul X, T_{ij}-->$_X T_{kl}$, dacă T_{ij}-->$^C{}_X T_{kl}$ și T_{ij} < T_{kl}.

(4) Tranzacția T_i precedă tranzacția T_k în planificarea S(τ), T_i-->T_k, dacă există un obiect X și o acțiune $T_{ij}\in T_i$ și $T_{kl}\in T_k$ astfel încît T_{ij}-->$_X T_{kl}$.

În aceste condiții, relația de precedență --> ordonează parțial setul de tranzacții .

Teorema:

O planificare S(τ) este *serializabila D* dacă și numai dacă relația sa de precedență este aciclică.

BUPT

Verificarea corectitudinii unei planificări $S(\tau)$ se reduce la a testa dacă graful relației de precedență --> este aciclic. Acest graf, denumit *graful de serializare D al unei planificări $S(\tau)$*, DSRG(S(τ)) se definește astfel:

Definiție:

Graful de serializare D al unei planificări $S(\tau)$, DSRG(S(τ)), este un graf orientat DSRG(S(τ))=(V,A) a cărei mulțime de noduri corespunde setului de tranzacții extinse cu tranzacția inițială și cu una finală, T_o și T_f și a cărei mulțime de arce este $A=\{(T_i,T_j) : T_i\text{-->}T_j,\ T_i,\ T_j \in \tau \cup \{T_o,T_f\}\}$.

Ordinea tranzacțiilor dintr-o planificare, implicată de relația --> se numește *ordine de serializare*; semnificația ei este următoarea: dacă tranzacțiile dintr-un set ar fi executate serial, în ordinea de serializare, atunci planificarea obținută în acest mod ar fi echivalentă cu planificarea $S(\tau)$. În fig. 7.4.2.8. se arată un exemplu de graf de serializare D, pentru setul de tranzacții din fig. 7.4.2.6., cu graful planificării în fig. 7.4.2.7. Ordinea de serializare a tranzacțiilor este T_o, T_2, T_1, T_3, T_f. Setul tuturor planificarilor unui set de tranzacții care satisface criteriul de serializare D, DSR, este inclus în SR.

Criteriul de serializare D este mai restrictiv decît criteriul de serializare prezentat anterior. Exemplul din figurile 7.4.2.9, 7.4.2.10, 7.4.2.11. demonstrează acest lucru. Se consideră planificarea din fig. 7.4.2.9. Graful de serializare construit pentru această planificare în figura 7.4.2.10. este aciclic, prin urmare planificarea este serializabila. Setul de arce se determină astfel: arcul (T_1,T_2) trebuie să existe în SRG(S(τ)), deoarece <T2, read_obiect$_1$(X,S$_1$), V$_1$> citește valoarea lui X scrisa de <T$_1$, write_obiect$_1$(X,S$_1$), V$_1$)>; arcul (T_2,T_f) trebuie să existe în SRG(S(τ)), deoarece <T$_f$, read_obiect$_1$(X,S$_1$), V$_1$> citește valoarea lui X scrisa de <T$_2$, write_obiect$_2$(X,S$_1$), V$_2$>. Datorită componenței planificării $S(\tau)$ trebuie inclus unul din următoarele două arce:(T_3,T_1) sau (T_2,T_3); fie (T_3,T_1) arcul inclus. În sfîrșit, conform condițiilor (3), (4), (5) din definiția grafului de serializare mai trebuie incluse în SRG(S(τ)) și următoarele arce: (T_0,T_1), (T_0,T_2), (T_3,T_2). Planificarea $S(\tau)$ este serializabilă, dar nu și serializabila D, deoarece graful DSRG(S(τ)) (fig. 7.4.2.11.) conține un ciclu.

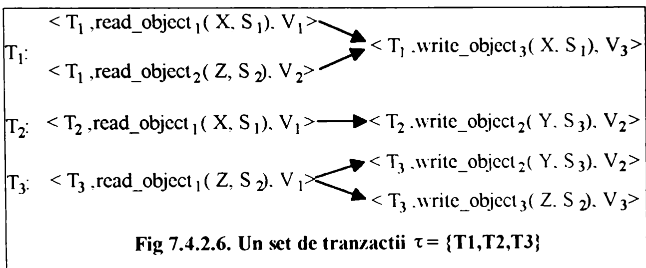


Fig 7.4.2.6. Un set de tranzactii $\tau = \{T1,T2,T3\}$

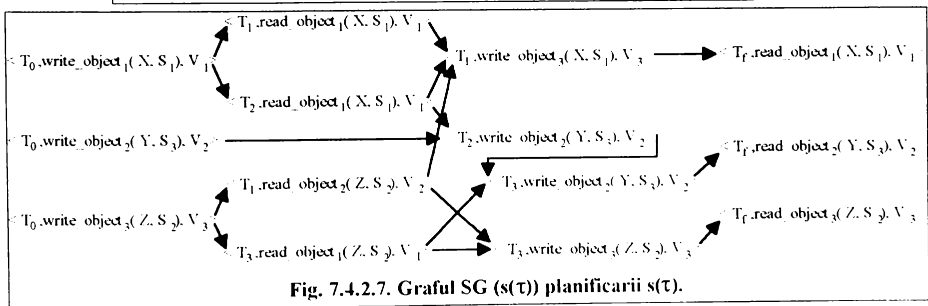


Fig. 7.4.2.7. Graful SG (s(τ)) planificarii s(τ).

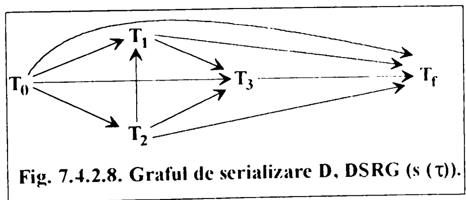


Fig. 7.4.2.8. Graful de serializare D, DSRG (s (τ)).

BUPT

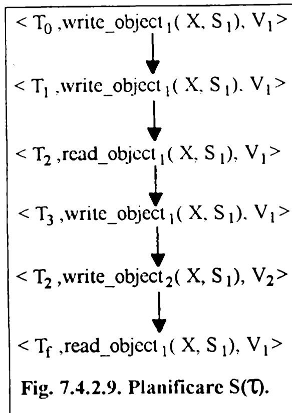


Fig. 7.4.2.9. Planificare S(T).

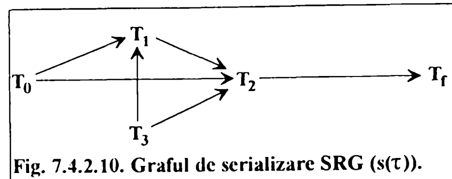


Fig. 7.4.2.10. Graful de serializare SRG (s(τ)).

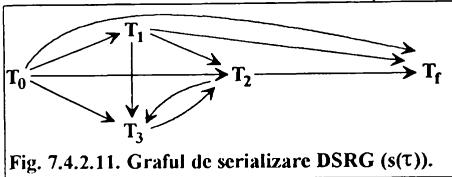


Fig. 7.4.2.11. Graful de serializare DSRG (s(τ)).

7.4.3. Mecanisme tradiționale pentru controlul concurenței

Problema consistenței datelor în aplicațiile tradiționale de baze de date a fost redusă la testarea serializabilității planificarilor; acest lucru se datorează faptului că nu se cunosc constrîngerile de consistență: fiecare acțiune într-o tranzacție se abstractizează printr-o operație *read* sau *write*. Problema obținerii serializabilității poate fi descompusă în două subprobleme: *sincronizarea read-write* și *sincronizarea write-write*, notate pe scurt sincronizările *rw* și *ww* ([BG81]). Sincronizarea *rw* se referă la serializarea tranzacțiilor în așa fel încît fiecare operație *read* citește aceeași valoare ca și cind tranzacțiile ar fi executate serial; sincronizarea ww se referă la serializarea tranzacțiilor în așa fel încît ultima operație *write* a fiecărei tranzacții lasă datele în aceeași stare ca și execuția seriala. Respectarea serializărior *rw* și *ww* generează o prelucrare consistentă. Astfel, deși sistemul de gestiune al bazei de date distribuite nu dispune de informații asupra constrîngerilor de consistență specifice aplicației, garantează consistența, permițînd numai execuții serializabile ale tranzacțiilor concurente. Conceptul de serializabilitate al tranzacțiilor este propriu mecanismelor tradiționale pentru controlul concurenței. Dacă sunt disponibile informații semantice despre tranzacții și operațiile lor pot fi permise și planificări neserializabile, dar totuși consistente (vezi secțiunea 7.4.7.2).

Mecanismele tradiționale de rezolvare ale problemelor controlului concurenței se împart în următoarele categorii: mecanisme bazate pe blocarea cu *lock*-uri, mecanisme bazate pe mărcile de timp și mecanisme pentru controlul optimist al concurenței. Unele mecanisme adaugă și alte concepte precum granularitatea blocarilor și tranzacții încuibate.

7.4.4. Mecanisme bazate pe blocarea cu *lock*-uri

Cele mai utilizate mecanisme pentru controlul concurenței sunt cele bazate pe *lock-uri*, acestea reprezentind o facilitate specială pentru controlul accesului la obiectele partajate. Utilizînd *lock*-uri, tranzacțiile pot bloca obiecte în scopul asigurării accesului exclusiv, pe durata unei stări temporare inconsistente. Aceasta implică faptul că alte tranzacții care necesită accesul la obiectele blocate, fie vor fi puse în așteptare, fie vor fi abortate, fie vor întrerupe tranzacția care deține obiectul blocat. Un obiect blocat și modificat de o tranzacție abortată sau întreruptă poate fi deblocat numai dupa ce starea obiectului este restaurată la cea de dinaintea începerii tranzacției.

Există trei tipuri de acțiuni relative la *lock*-uri: (1) < T. lock_X<E.S> > specifică o cerere a tranzacției T pentru un *lock* exclusiv la obiectul <E,S>; cererea este satisfacută numai dacă nici o altă tranzacție nu deține un *lock* de orice tip asupra obiectului; (2) < T. lock_S <E,S> > specifică o cerere a tranzacției T pentru un *lock* partajat la obiectul <E,S>; cererea este satisfacută numai dacă nici o altă tranzacție nu deține un *lock* exclusiv asupra obiectului; (3) < T, unlock <E,S> > specifică eliberarea *lock*-ului deiînut asupra obiectului <E,S>.

Două *lock*-uri sunt *în conflict* dacă implică același obiect și cel puțin unul dintre ele este un *lock* exclusiv. Unei tranzacții i se permite să obțină un *lock* asupra unui obiect, de mai multe ori în moduri diferite: modul exclusiv de blocare este menținut, însă, pînă la o cerere unlock.

Avantajele principale ale metodelor bazate pe *lock*-uri sunt: (1) ordinea de serializare nu este stabilită a priori, ci depinde de progresul execuțiilor tranzacțiilor; (2) sunt disponibili algoritmi ierarhici eficienți. Dezavantajele sunt legate de (1) accesul la obiecte este restrictiv, datorită *lock*-urilor; gradul de concurență este mai mic decît într-o serializare D; (2) pot apare interblocări.

BUPT

7.4.4.1. Mecanismul blocării în două faze (2PL)

Mecanismul blocării în două faze , introdus în [EGT78] este astăzi unanim acceptat ca soluţie standard pentru controlul concurenţei în bazele de date conventionale.

Mecanismul impune următoarele constrîngeri ale manierei în care tranzacţiile sunt structurate: (1) tranzacţia este *corect structurata,* adică citeşte un obiect numai cît timp deţine un *lock* exclusiv sau partajat asupra obiectului şi scrie un obiect numai dacă deţine un *lock* exclusiv asupra obiectului; (2) tranzacţia este *în două faze*, adică nu execută o acţiune *lock* dupa ce a executat o acţiune unlock. Prima fază a tranzacţiei, denumită *fază de acumulare,* începe cu prima acţiune şi se termină la prima acţiune unlock. Cea de-a doua faza, denumită *faza de restrangere*, începe cu prima acţiune unlock şi continua pînă la ultima acţiune.

Execuţia concurentă a unui set de tranzacţii generează execuţia unei secvente de acţiuni intermixate asupra obiectelor din sistem. Fiecare planificare dintr-un nod constituie o planificare locală, iar controlul structurii ei este aspectul fundamental al mecanismului de control al concurenţei. O planificare este *legală* dacă în cadrul ei tranzacţiile nu contin *lock*-uri în conflict. Următoarele două teoreme stabilesc relaţia dintre structura tranzacţiilor şi structura planificărilor locale şi planificărilor globale.

Teorema 1: (pentru un sistem centralizat)
Dacă $\{T_1,T_2,...,T_k\}$ este un set de tranzacţii corect structurate şi în două faze, atunci orice planificare Q_0 este consistentă.

Teorema 2: (pentru un sistem distribuit)
Fie $Q_1,...,Q_n$ un set de planificări locale ale tranzacţiilor $\{T_1,T_2,...,T_m\}$. Atunci există o planificare Q astfel încît: (a) secvenţele de acţiuni din T_i şi Q_i sunt subsecvente ale lui Q; (b) dacă fiecare planificare Q_i este legală, atunci şi Q este legală.

Demonstratie:
Deoarece aceasta teoremă stabileşte că execuţia acţiunilor în planificările locale poate fi considerată ca execuţia unei singure planificări într-un nod centralizat se propune următoarea demonstraţie. Se consideră că fiecare nod j este dotat cu un ceas (imaginar) implementat printr-un simplu contor, a cărui valoare este desemnată prin C_j. Presupunem că fiecare manager de tranzacţie (TM) menţine de asemenea un ceas desemnat prin C_T. Dacă TM este în nodul i, atunci $C_T=C_i$. Cînd MT adaugă o cerere pentru o acţiune la obiectul din nodul j, incrementează mai întîi C_i şi apoi trimite un mesaj: ("REQUEST", C_j, T, A <0>, V) la nodul j. La primirea mesajului, nodul j actualizează ceasul sau $Cj=max(C_j, C_i)$. Cînd nodul j execută acţiunea A asupra obiectului O pentru tranzacţia T, incrementează C_j şi trimite un mesaj de confirmare la TM, ("ACK", C_j, T, A <0>, V). La primirea mesajului de confirmare, TM stabileste: $C_i=max(C_i,C_j)$ şi $C_T=C_i$. Pentru a forma o planificare globală echivalentă, se pot combina planificările locale, sortînd acţiunile în ordine crescatoare dupa (C_i, număr nod). Aceasta planificare globală satisface proprietăţile a) şi b) şi teorema este demonstrata.

Un exemplu de planificare legală şi serială a tranzacţiilor din fig. 7.2.1.1. este prezentat în fig. 7.4.4.1.1.

Implementarea unui mecanism de control al concurenţei bazate pe *lock*-uri într-un sistem distribuit poate fi facută dacă se menţine în fiecare nod un manager al *lock*-urilor, căruia se adresează acţiunile relative la *lock*-uri pentru obiectele de pe acel nod. În fig. 7.4.4.1.2. se arată un exemplu în acest sens: în nodul 1 sunt două obiecte A şi B, iar în nodul 2 obiectul C. O tranzacţie T va adresa acţiunile *lock* pentru obiectele A şi B managerului de *lock*-uri local, iar cererile *lock* pentru obiectul C managerului de *lock*-uri de pe nodul 2. Managerul de *lock*-uri poate fi parte integrantă a managerului de obiecte.

Se apreciază că algoritmul 2PL este optimal, în sensul că asigură un grad mai mare de concurenţă decît alte metode dacă: (1) obiectele sistemului distribuit sunt independente şi reprezintă acelaşi nivel de abstractizare; (2) tranzacţiile sunt dependente de date.

BUPT

```
< T1, lock_X (A, S) >
< T1, read_object (A,S), BalantaA >
< T1, lock_X (B, S) >
< T1, read_object (B, S), BalantaB >
< T1, write_object (A, S), BalantaA-100 000 >
< T1, write_object (B, S), BalantaB+100 000 >
< T1, unlock (A, S) >
< T1, unlock (B, S) >
< T2, lock_X (B, S) >
< T2, read_object (B,S), BalantaB >
< T2, lock_X (C, S) >
< T2, read_object (C,S), BalantaC >
< T2, write_object (B, S), BalantaB-200 000>
< T2, write_object (C, S), BalantaC+200 000>
< T2, unlock (B, S) >
< T2, unlock (C, S) >
< T3, lock_S (A, S) >
< T3, read_object (A, S), BalantaA >
< T3, lock_S (B, S) >
< T3, read_object (B, S), BalantaB >
print BalantaA
print BalantaB
< T3, unlock (A, S) >
< T3, unlock (A, S) >
```

Fig. 7.4.4.1.1. O planificare legala si seriala

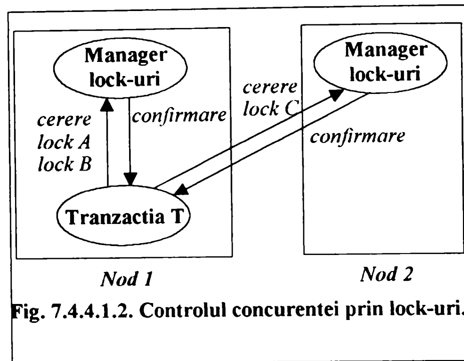


Fig. 7.4.4.1.2. Controlul concurentei prin lock-uri.

7.4.4.2 Algoritmi de blocare ierarhici

Algoritmii de blocare ierarhici pun problema granularităţii datelor blocate: aceasta constă în a selecta dimensiunea optimală a unităţii de blocare. Problema generează acceptarea unui compromis între maximizarea gradului de concurenţă şi minimizarea regiei de sistem necesară pentru blocare. Pentru a maximiza gradul de concurenţă pentru tranzacţii mici, sunt adecvate multe unităţi de blocare mici; pentru a minimiza regia de sistem pentru tranzacţii mari, sunt adecvate puţine unităţi de blocare dar mari. Conceptul de granularitate a fost propus pentru a încapsula ideea unor unităţi de blocare ierarhice. O ierarhie de granularităţi poate fi reprezentată printr-un graf aciclic ale cărui noduri sunt granularităţile şi ale cărui arce conectează granularităţile de nivel înalt cu granularităţile de nivel jos; un exemplu este prezentat în figura 7.4.4.2.1.

Algoritmii de blocare ierarhici utilizează două tipuri de moduri de blocare: *modul de bază* (prezentat anterior la 7.4.4.1) şi *modul intenţiilor de blocare*. Două moduri de blocare sunt *compatibile* dacă permit accesul concurent la acelaşi obiect. Pentru a bloca un obiect de la un anumit nivel, trebuie obţinută mai întîi permisiunea pentru intenţia de acces de la toate componentele ascendente de pe nivelele superioare; de exemplu, dacă un obiect A are deja asociat *lock*-ul de intenţie I, atunci o cerere de blocare a lui A în modul S (shared) sau X (exclusiv) va fi nesatisfacută deoarece s-ar putea ca A să aibă o componentă de nivel mai jos deja blocată în modul X; astfel, modurile X, S şi I sunt incompatibile unul cu altul.

Pentru a permite însă un grad mai mare de concurenţă, s-au definit trei moduri de intenţii de acces la un obiect: *intenţia de acces in mod partajat* (desemnată prin IS), *intenţia de acces în mod exclusiv* (desemnată prin IX) şi *accesul mixt* (desemnat prin SIX). Cele cinci moduri de blocare pot fi rezumate astfel: (1) modul IS asupra unui obiect specifică faptul că tranzacţia doreşte să citească componentele descendente ale obiectului (permite blocarea partajată a acestora în modurile S sau IS); (2) modul IX asupra unui obiect specifică faptul că tranzacţia doreşte să scrie şi/sau să citească componentele descendente ale obiectului; (3) modul S asupra unui obiect specifică faptul că tranzacţiei i se permite accesul în mod partajat la obiectul respectiv şi implicit la toate componentele descendente; (4) modul SIX asupra unui obiect specifică faptul că tranzacţia va deţine modul S şi IX asupra obiectului (poate citi obiectul) şi că are dreptul să scrie şi/sau citească descendenţii săi; (5) modul X asupra unui obiect specifică faptul că tranzacţia va avea acces exclusiv asupra obiectului şi, implicit, acces exclusiv şi la componentele descendente. În tabelul din fig.7.4.4.4.2 se prezintă compatibilitatea dintre cele 5 moduri.

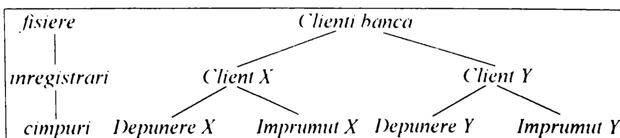


Fig. 7.4.4.2.1. O ierarhie de lock-uri.

BUPT

mod de blocare curent / mod de blocare cerut	IS	IX	S	SIX	X
IS	***DA***	***DA***	***DA***	***DA***	*NU*
IX	***DA***	***DA***	NU	NU	NU
S	***DA***	NU	***DA***	NU	NU
SIX	***DA***	NU	NU	NU	NU
X	NU	NU	NU	NU	NU

Fig. 7.4.4.2.2 Compatibilitatea modurilor de blocare pentru algoritmi de blocare ierarhici.

Pentru a obţine corect accesul la o componentă obiect, o tranzacţie trebuie să se conformeze următorului algoritm:

Algoritm:

(1) Pentru a obţine accesul S sau IS la o componentă a unui obiect, o tranzacţie trebuie se deţină un *lock* IS sau IX asupra tuturor componentelor predecesoare; (2) Pentru a obţine accesul X, SIX sau IX la o componentă a unui obiect, o tranzacţie trebuie să deţină un *lock* SIX sau IX asupra componentelor predecesoare; (3) Toate *lock*-urile trebuie eliberate fie la sfîrsitul execuţiei tranzacţiei, fie în ordinea frunze catre rădăcină în timpul execuţiei tranzacţiei.

În figura 7.4.4.2.3. se prezintă ca exemplu două tranzacţii care actualizează cîmpurile "Depunere Y", utilizînd corect *lock*-uri ierarhice.

Se poate demonstra că orice planificare concurentă $S(\tau)$ a tranzacţiilor τ, care respectă algoritmul de mai sus, este serializabilă; în figura 7.4.4.2.4 se prezintă un exemplu de planificare serializabilă a tranzacţiilor din figura 7.4.4.2.3.(în exemplu s-au omis operaţiile de scriere şi citire).

Algoritmul aratat prezintă însă pericolul apariţiei interblocării. Un exemplu de planificare a tranzacţiilor T_1 şi T_2 în care apare interblocare, este prezentat în figura 7.4.4.2.5.

T1:
<T1.lock_IX. "Clienti_banca">
<T1.lock_X. "Client X">
<T1.lock_S. "Client Y">
<T1.read "Depunere X">
<T1.read "Depunere Y">
<T1.read "Imprumut Y">
<T1.write "Imprumut X">
<T1.unlock "ClientY">
<T1.unlock "Client X">
<T1.unlock "Clienti_banca">

T2:
<T2.lock_IX. "Clienti_banca">
<T2.lock_IX. "Client Y">
<T2.lock_X. "Depunere Y">
<T2.read "Depunere Y">
<T2.lock_IS. "Client X">
<T2.lock_S. "Depunere X">
<T2.read "Depunere X">
<T2.write "Depunere Y">
T2.unlock "Depunere X">
<T2.unlock "Client X">
<T2.unlock "Depunere Y">
<T2.unlock "Client Y">
<T2.unlock "Clienti_banca">

Fig. 7.4.4.2.3.Tranzactii care utilizeaza lock-uri ierarhice

S(t):
<T1. lock_IX. "Clienti_banca">
<T2. lock_IX. "Clienti_banca">
<T1. lock_X. "Client_X">
<T1. lock_S. "Client_Y">
<T1. unlock. "Client_Y" >
<T2. lock_IX. "Client_Y">
<T2. lock_X. "Depunere Y">
<T1. unlock. "Client_X">
<T2. lock_IS. "Client_X>
<T2. lock_S. "Depunere X">
<T1. unlock. "Clienti_banca">
<T2. unlock. "Depunere X">
<T2. unlock. "Client_X">
<T2. unlock. "Depunere Y">
<T2. unlock. "Client_Y">
<T2. unlock. "Clienti_banca">

Fig.7.4.4.2.4 Planificare seriabila a tranzactiilor T1 si T2

S(t)::
<T1. lock_IX. "Clienti_banca">
<T2. lock_IX. "Clienti_banca">
<T1. lock_X. "Client_X">
<T2. lock_IX. "Client_Y">
<T2. lock_X. "Depunere Y">
<T1. lock_S. "Client_Y">
<T2. lock_IS. "Client_X">
.... interblocare

Fig.7.4.4.2.5.. Planificare ce genereaza o interblocare

BUPT

7.4.4.3 Grafuri aciclice de lock-uri

Conceptul de lock-uri ierarhice poate fi extins și la un set de obiecte organizate într-un graf orientat aciclic, în care componentă a unui obiect poate avea mai mult decît un părinte; un exemplu este prezentat în fig. 7.4.4.3. În acest model, se consideră următoarele extensii asupra *lock*-urilor: (1) O tranzacție deține implicit un *lock* S, dacă a obținut, implicit sau explicit un *lock* S, SIX sau X la cel puțin unul dintre părinții componentei; aceasta implică faptul că tranzacția a obținut explicit un acces S, SIX sau X la unul dintre predecesori. (2) O tranzacție deține implicit un *lock* X asupra unei componente C a unei resurse dacă a obținut implicit un *lock* X asupra tuturor părinților; aceasta înseamnă că pentru o componentă în fiecare cale care conduce la componentă C s-a obținut explicit modul X și pentru toți predecesorii săi s-a obținut modul IX sau SIX. Pentru a obține accesul, fiecare tranzacție trebuie să respecte următorul protocol: (1) Pentru a obține un *lock* S sau IS, o tranzacție trebuie să dețină un *lock* IS (sau un *lock* mai puternic) la cel puțin unul dintre părinți și la toți predecesorii săi dintr-o cale catre rădăcină. (2) Pentru a obține un *lock* IX, SIX sau X la o componentă a unui obiect, o tranzacție trebuie să dețină un *lock* IX (sau un *lock* mai puternic) pentru toți părinții sai. Aceasta implică faptul că tranzacția va deține un *lock* IX (sau unul mai puternic) pentru toți predecesorii acestor componente și nici o altă tranzacție nu va putea deține un *lock* asupra unei componente predecesoare care este incompatibil cu IX. (3) cînd o tranzacție se termină, *lock*-urile pot fi eliberate în orice ordine; în caz cotrar, ele trebuie eliberate în ordinea frunza-rădăcină. În figura 7.4.4.3.b) se prezintă un exemplu de obținere a unui acces S la înregistrările dintr-un fișier F, utilizînd indexul I.

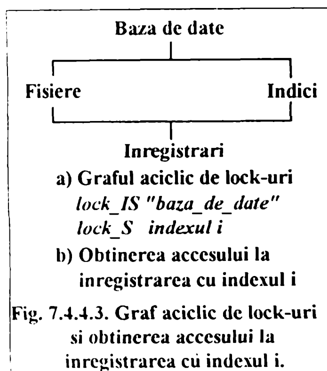


Fig. 7.4.4.3. Graf aciclic de lock-uri si obtinerea accesului la inregistrarea cu indexul i.

7.4.5 Mecanisme bazate pe marca de timp

Una dintre problemele mecanismelor de control al concurenței este posibilitatea apariției interblocărilor. Problema interblocărilor poate fi rezolvată dacă se asociază fiecărei tranzacții o marcă de timp. Mecanismele de control al concurenței din această secțiune se bazează toate pe marca de timp.

7.4.5.1 Concepte de bază

Marca de timp asociată unei tranzacții, TS(T$_i$) este un număr obținut prin concatenarea valorii ceasului (logic sau fizic) cu identificatorul nodului de la care a fost inițiată tranzacția. Pentru fiecare obiect al sistemului distribuit de obiecte se asociază două mărci de timp: *marca de timp pentru citire* $TS_R(x)$ și *marca de timp pentru scriere* $TS_W(x)$; acestea se setează la valoarea mărcii de timp asociată ultimei tranzacții care a citit, respectiv a scris obiectul. Mecanismele de control al concurenței bazate pe marca de timp presupun că ordinea de serializare a unei planificări concurente a tranzacțiilor este strict determinată prin valorile mărcilor de timp ale tranzacțiilor. Algoritmii de control bazați pe marca de timp trebuie să verifice că această ordine nu este violată în timpul execuției concurente a tranzacțiilor (datorită întîrzierilor de comunicație, de exemplu). Ordinea de serializare hotarîtă a priori este principalul dezavantaj al metodelor bazate pe marca de timp, deoarece orice violare a acestei ordini conduce la abortări multiple ale tranzacțiilor ale căror cereri de acces sosesc prea tîrziu; de aceea au fost propuși diverși algoritmi al căror scop era minimizarea efectelor acestui dezavantaj.

Algoritmii bazați pe marca de timp pot fi grupați în trei categorii de bază: deterministi, nedeterminiști și semi-determiniști.

Algoritmii determiniști includ algoritmii în care managerii de obiecte permit accesul la obiecte numai dacă acesta nu provoacă restartarea unei tranzacții; cu alte cuvinte, o cerere de acces la un obiect de catre o tranzacție este buferată de un manager de obiecte pînă nu mai există tranzacții cu mărci de timp mai mici care cer acces la același obiect. În această categorie se includ *algoritmii conservativi* ([CP84], [BG81], [CG87]).

Algoritmii nedeterminiști se consideră descendenți direcți ai procedurii *two-phase commit*. Cînd o tranzacție T_i cere un acces la un obiect in conflict cu accesul la același obiect garantat unei alte tranzacții T_j, $TS(T_j)>TS(T_i)$, accesul la obiect al lui T_i depinde de starea tranzacției T_j. Dacă cererea tranzacției T_j a fost acceptată, atunci cererea lui T_i va fi rejectată, în caz contrar, cererea lui T_i este acceptată, provocînd rejectarea cererii lui T_j.

Algoritmii semi-determiniști includ algoritmii care utilizează o coadă de tranzacții, în scopul minimizării numărului de abortări. Fiecărui obiect i se asociază o coadă de tranzacții care așteaptă accesul la un obiect. Un manager de obiecte, după execuția cererii de acces curente, preia tranzacția cu cea mai mică marca de timp din coadă. În cazul în care coada asociată unui obiect este vidă în momentul sosirii unei cereri, atunci cererea este acceptată imediat cu condiția ca tranzacția de la care provine cererea să aibă o marca de timp mai mare decît tranzacția care a lansat ultima cerere de acces executată.

Avantajele metodelor bazate pe mărcile de timp sunt: (1) O tranzacție poate actualiza un obiect chiar imediat după ce altă tranzacție a citit-o; actualizările pot fi executate concurent atîta timp cît ordinea mărcilor de timp nu este violată. Astfel gradul de concurență este îmbunătățit. (2) Interblocările sunt eliminate. Dezavantajul principal este faptul ca ordinea de serializare hotărîtă apriori conduce la abortarea și restartarea tranzacțiilor care încearcă să acceseze obiecte în ordinea inversă a mărcilor de timp. Minimizarea restartarilor la infinit este problema crucială.

BUPT

7.4.5.2 Algoritmul fundamental pentru controlul concurenţei pe baza mărcii de timp

Algoritmul fundamental în controlul concurenţei pe baza mărcii de timp poate fi descris, într-o primă fază, ignorînd respectarea atomicitatii, astfel:

Algoritm:
1. Managerul de tranzacţii asociază o marcaă de timp unică fiecărei noi tranzacţii T.
2. Managerul de tranzacţii asociază marca de timp tuturor acţiunilor trimise pentru execuţie managerului de obiecte, în cadrul acelei tranzacţii.
3. Fiecare manager de obiecte foloseşte un planificator pe bază de marcă de timp care planifică acţiunile locale corespunzător ordinei mărcilor de timp. Pentru fiecare obiect O, planificatorul înregistrează cea mai mare marcă de timp a tuturor acţiunilor *read object* terminate şi a tuturor acţiunilor *write_object* terminate. Aceste mărci de timp vor fi notate cu TS_R şi TS_W.
4. Pentru o acţiune *read_object*, planificatorul compară marca sa de timp TS cu TS_W. Dacă $TS<TS_W$ planificatorul rejectează acţiunea iar managerul de tranzacţii abortează tranzacţia; în caz cotrar, planificatorul plasează acţiunea într-o coada FIFO cu acţiuni în aşteptare la obiectul O şi setează TS_R la $\max(TS_R,TS)$.
5. Pentru o acţiune *writeÎobject*, planificatorul compară marca sa de timp TS cu maximum dintre TS_R şi TS_W. Planificatorul rejectează acţiunea dacă TS este mai mica; în caz contrar, plasează acţiunea în coada şi setează TS_W la TS.
6. Dacă este necesar să se aborteze o tranzacţie managerul de tranzacţii ii va asigna o noua marcă de timp (mai mare) şi o va restarta.

Algoritmul de bază introduce însă trei probleme specifice: ***restartări ciclice, restartări nenecesare*** şi ***restartări la infinit***. Pentru a prezenta *restartările ciclice* se consideră următoarea situaţie: două tranzacţii T_1 şi T_2 fiecare constind intr-o acţiune *read_object(S,O)* urmată de o acţiune *write_object(S,O)* relative la acelaşi obiect O sunt executate de următoarea planificare:

... <T_1, read_object(S,O), V_1>.... <T_2, read_object(S,O), V_1>,
<T_1, write_object(S,O), V_2>, <T_2, write_object(S,O), V_2>...

Dacă marca de timp a lui T_1 este mai mică decît cea a lui T_2 atunci <T_1, write_object(S,O), V_3> va fi rejectată iar T_1 restartată cu o marcă de timp mai mare. Dacă imediat T_1 execută o noua acţiune read_object(S,O) atunci <T_2, write_object(S,O), V_2> va fi rejectată iar T_2 restartată. Apare astfel o secvenţă de restartări ale tranzacţiilor T_1 şi T_2.

Restartările nenecesare apar dacă secvenţa de execuţie a acţiunilor este în ordinea inversă a mărcilor de timp. Un exemplu în acest sens îl constituie cazul unei tranzacţii T_2 a cărei marcă de timp este mai mare decît cea a lui T_1, dar toate acţiunile lui T_2 sunt terminate complet înaintea oricărei acţiuni a lui T_1; T_1 va fi rejectată chiar dacă planificarea serială în care T_2 este urmată de T_1 este perfect acceptabilă.

Restartările la infinit apar cînd o tranzacţie mai veche, la fiecare restartare, intră în conflict cu o tranzacţie mai nouă; aceasta este o problema de amînare la infinit.

Pentru a asigura atomicitatea tranzacţiilor este necesar să se integreze procedura de mai sus cu procedura *two-phase commit* care permite abortarea sau realizarea tuturor operaţiilor unei tranzacţii (în principal, contează operaţiile *write*). În acest caz, procedura *two-phase commit* se realizează pe baza utilizării unor operaţii *prewrite*, care sunt executate de tranzacţii în locul operaţiilor *write*. Operaţiile *prewrite* sunt buferate în managerii de obiecte şi nu se aplică obiectelor locale; buferarea unei operaţii înseamnă inregistrarea ei intr-un bufer pentru o execuţie ulterioară. Numai cînd toate operaţiile *prewrite* ale unei tranzacţii sunt acceptate, tranzacţia poate fi realizată şi efectuate şi operaţiile *write*. Dacă toate operaţiile *prewrite* ale unei tranzacţii au fost acceptate, operaţiile *write* corespondente nu pot fi rejectate.

Scopul procedurii *two-phase commit* este de a evita actualizările partiale provocate, de exemplu de avaria unui nod. În plus, procedura *two-phase commit* previne aşa numitele ***abortări in cascadă*** sau *efectul domino*; acestea au loc cînd abortarea unei tranzacţii parţial comisă provoacă abortarea altor tranzacţii parţial comise care au citit valori scrise de tranzacţia abortată, efectul putîndu-se propaga în continuare la alte tranzacţii.

În descrierea algoritmului fundamental pentru controlul concurenţei pe baza mărcii de timp, se vor folosi următoarele variabile: TS_R marca de timp maximă a acţiunilor *read-object* executate, TS_W marca de timp maximă a acţiunilor *write-object*, min_TS_R marca de timp minimă a oricărei acţiuni *read-object* bufferate, min_TS_W = marca de timp minimă a oricărei acţiuni *write-object* bufferate, min_TS_P marca de timp minimă a oricărei acţiuni *prewrite* bufferate. Se presupune că: (1) toate aceste variabile, ca şi mărcile de timp generate de managerii de tranzacţii sunt iniţializate cu zero; (2) min_TS_R, min_TS_W şi min_TS_P reţin valorile cînd bufferele corespondente devin vide; (3) fiecare acţiune *read-object*, *write-object* şi *prewrite* este o operaţie indivizibila raportată la planificatorul local.
În aceste condiţii, algoritmul poate fi descris astfel:

BUPT

<T_i , read-object (X, S), V> ::

if $TS(T_i) < TS_W \rightarrow$

<rejecteaza cererea, aborteaza tranzactia si restarteaza tranzactia cu o noua marca de timp>

[] $TS(T_i) \geq TS_W \rightarrow$

if $TS(T_i) > min_TS_P(X) \rightarrow$

<plaseaza cererea de citire in buffer>

[] $TS(T_i) \leq min_TS_P(X) \rightarrow$

< read_X >

$TS_R(X) := max(TS(T_i), TS_R(X))$

fi

fi

<T_i , write-object (X, S), V> ::

if $TS(T_i) > min_TS_R(X)$ *or* $TS(T_i) > min_TS_P(X) \rightarrow$

plaseaza cererea de scriere in buffer

[] $TS(T_i) \leq min_TS_R(X)$ *and* $TS(T_i) \leq min_TS_P(X) \rightarrow$

< realizeaza_write (Ti, X) >

$TS_W(X) := TS(T_i)$

fi

<T_i , prewrite (X, S), V> ::

if $TS(T_i) < min_TS_R(X)$ *or* $TS(T_i) < min_TS_W(X) \rightarrow$

<rejecteaza cererea, aborteaza si restarteaza T_i tranzactia cu o noua marca de timp >

[] $TS(T_i) \geq min_TS_R(X)$ *and* $TS(T_i) \geq min_TS_W(X) \rightarrow$

< plaseaza cererea prewrite in buffer >

fi

Procedura *realizeaza_write(T_i, X)* realizează actualizarea obiectului și înlătură din buffer cererea prewrite. Dacă valoarea $min_TS_P(X)$ crește după această ștergere, atunci inițiază o procedură care testează buffer-ul și determină dacă noua valoare $min_TS_P(X)$ nu deblochează o cerere ***read*** sau ***write*** a altei tranzacții, fie ea T_j, pentru care este adevarată acum condiția $TS(T_j) < min_TS_P(X)$. Deblocarea acestor cereri poate provoca actualizarea din nou a variabilelor $min_TS_R(X)$ și $min_TS_P(X)$, și, în consecință, testarea din nou a buffer-ului.

7.4.5.3. Optimizări ale algoritmului de bază

În cele ce urmează sunt descrise trei tehnici de optimizare ale algoritmului de bază ([BK91]):

1. Regula lui Thomas

Dacă două cereri *write-object* consecutive sunt în conflict și cererea mai nouă are marca de timp mai mică decît cealaltă, atunci cea mai nouă poate fi ignorată, eliminînd astfel acțiunile *write_object* nenecesare.

2. Folosirea mai multor versiuni ale unui obiect

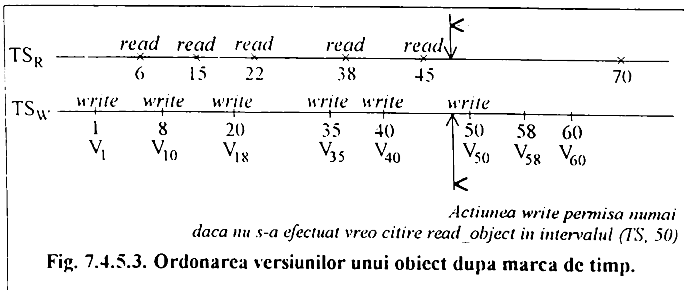
Prin folosirea mai multor versiuni pentru un obiect, cu costul irosirii de spațiu, se evită rejectarea acțiunilor *read-object*. În acest caz un obiect O constă într-o secvență de versiuni, fiecare de forma <TS_W, valoare >, care înregistrează mărcile de timp și valorile conținute în urma execuției cererilor *write-object* <O,S> și o secvență de mărci de timp TS_R pentru toate acțiunile *read-object* <O,S>. Crearea unor copii multiple ale unui obiect este arătată în fig. 7.4.5.3. Pentru a se reduce cerințele de memorie, versiunile mai vechi ale unui obiect sunt descărcate.

Implementarea acestei tehnici se face ținînd cont de următoarele reguli:

(1) O acțiune *read-object* cu marca de timp TS se execută prin citirea versiunii obiectului O cu cea mai mare marcă de timp mai mică decît TS și adaugînd marca TS setului de mărci TS_R.

(2) O acțiune *write-object* cu marca de timp TS se execută dacă nu există TS_R în intervalul semideschis [TS, cea mai mică TS_W > TS). În caz contrar se crează o nouă versiune a obiectului O cu marca de timp TS.

3. Algoritmi Wait-Die și Wound-Wait



Actiunea write permisa numai daca nu s-a efectuat vreo citire read_object in intervalul (TS, 50)

Fig. 7.4.5.3. Ordonarea versiunilor unui obiect dupa marca de timp.

Algoritmii Wait-Die și Wound-Wait se utilizează în conjuncție și cu prevenirea interblocării, fiind prezentați de aceea mai pe larg în secțiunea 7.5.2. Pe scurt, protocolul *wait-die*, o tranzacție T_i poate aștepta dupa T_j numai dacă T_i este mai veche (marca de timp mai mica); altfel este abortată și apoi restartată. În protocolul *wound-die*, o tranzacție T_i poate aștepta dupa T_j numai dacă T_i este mai nouă (cu o marcă de timp mai mare)

BUPT

7.4.5.4. Algoritmul conservativ de ordonare a mărcilor de timp

Algoritmul conservativ de ordonare al mărcilor de timp a fost introdus pentru a reduce numărul abortărilor din algoritmul de bază (CP84], [CG87]). În cadrul acestui algoritm o cerere de acces este acceptată numai dacă nu provoacă restartarea altei tranzacţii din sistem. În scopul eliminării restartărilor, operaţiile tranzacţiilor mai recente sunt bufferate pînă la terminarea execuţiilor unor tranzacţii mai vechi, cu care sunt în conflict. Aceasta presupune că fiecare nod comunică cu celelalte în scopul de a culege informaţii despre setul de tranzacţii curent procesate.

Fiecare manager de obiecte are asociat un set de 2*N cozi, care conţin cererile de acces de la tranzacţiile iniţiate de la N manageri de tranzacţii: TM_1, ..., TM_N. Fiecare manager de obiecte conţine o coadă *Rqueue(TM_l)* de cereri de acces ***read*** şi o coada *Wqueue (TM_l)* de cereri de acces *write* pentru fiecare modul TM_l. Fiecare cerere de acces sosită la un modul OM_k de la nodul TM_j este plasată în *RQueue(TM_j)* sau *Wqueue(TM_j)*. Se presupune că toate cererile de acces sosesc de la modulul TM_j la modulul OM_k în ordinea mărcilor de timp; pentru aceasta sunt necesare două condiţii: (1) ordinea mărcilor de timp nu trebuie să fie modificată de mediul de comunicaţie; (2) fiecare modul TM trebuie să trimită cererile de acces în ordinea mărcilor de timp. Aceste cerinţe pot fi îndeplinite în două moduri. Primul este de a executa tranzacţiile în ordinea strict serială, la fiecare modul TM. Aceasta micşorează gradul de concurenţă al sistemului la N. Un al doilea mod este de a executa toate tranzacţiile dintr-un modul TM_l în două etape. Toate cererile de citire se generează în prima etapă şi toate cererile de scriere în etapa a doua; aceasta asigură execuţia concurentă la nivelul unui modul TM_l.

Se prezintă în continuare procedurile de citire şi scriere pentru algoritmul conservativ. Pentru simplificare, se presupune că, atunci cînd un modul OM_K primeşte o cerere de scriere sau citire, fiecare coadă *RQueue (TM_l)* şi *WQueue (TM_l)*, l=1,2,...,N conţine cel puţin o cerere de acces la obiecte; deoarece s-a presupus că toată cererile de acces sosesc la OM_K în ordinea mărcilor de timp, *RQueue* şi *WQueue* conţin întotdeauna cele mai vechi cereri de acces netratate. Cu alte cuvinte, un modul OM_K nu poate primi o cerere de acces de la un manager de tranzacţii TM_l astfel încît marca de timp a acestei tranzacţii să fie mai mică decît marca de timp a tranzacţiilor în aşteptare în *RQueue (TM_l)* şi *WQueue(TM_l)*. Procedurile de citire şi scriere pentru algoritmul conservativ sunt următoarele:

<T_i,read_object(X,S),V>::
test = *true*
fa TM_K=TM_1, TM_N →
fa $T_j \in$ WQueue(TM_K) →
if $TS(T_j) < TS(T_i)$ → test = *false fi*
af
af
if test → <read X>
[] *not* test --> <plaseaza cererea read de la T_i in Rqueue(TM_l)
fi

<T_i,write_object(X,S),V>::
test = *true*
fa TM_K=TM_1, TM_N →
fa $T_j \in$ WQueue(TM_K) →
if $TS(T_j) < TS(T_i)$ → test = *false fi*
fa
fa $T_j \in$ RQueue(TM_K) → *if* $TS(T_j) < TS(T_i)$ → test = *false fi*
af
af
if test → < read (X) >
[]not test → <plaseaza cererea write de la T_i in Wqueue(TM_l)
fi

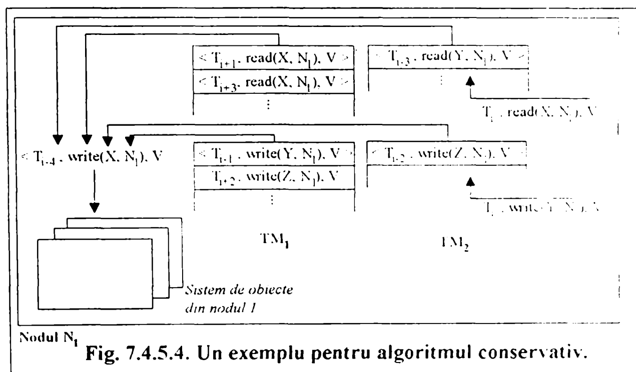


Fig. 7.4.5.4. Un exemplu pentru algoritmul conservativ.

În algoritmul de mai sus s-a presupus că atunci cînd un modul OM_K primeşte o cerere de acces la date, nici una din cele 2*N cozi ale sale, corespunzătoare modulelelor TM_1,...,TM_N nu este vidă; în caz contrar, ultima cerere dintr-o coadă ar trebui întîrziată pînă la sosirea unei noi cereri. Problema cozilor vide poate fi însă soluţionată şi astfel: fiecare modul TM, transmite periodic acţiuni nule fiecărui modul OM; o acţiune nulă nu specifică nici o operaţie asupra obiectelor, dar, fiind însoţită de marca de timp, rolul său este de a furniza ultima marcă de timp generată de un modul TM. O acţiune nulă poate fi transmisă periodic sau la cererea directă a unui modul OM.

În fig. 7.4.5.4. se prezintă un exemplu de utilizare al algoritmului conservativ. Sistemul este format din două noduri N_1 şi N_2, fiecare conţinînd cîte un modul OM şi TM; în figură se consideră execuţia cererilor de acces la obiectul din N_1. Se presupune că mărcile de timp ale tranzacţiilor sunt egale cu indicii lor. Figura arată starea cozilor *RQueue* şi *WQueue* ale modulului OM_1 cînd se primeşte o cerere de citire <T_i, read(X, N_1), V > în momentul prelucrării cererii <T_{i-4}, write(X, N_1), V >. Conform algoritmului conservativ, planificarea pentru execuţie a cererilor este următoarea: ...<T_{i-4}, write(X,N_1),V>, <T_{i-1}, write(Y,N_1),V>, <T_{i-2}, write(Z,N_1),V >, <T_{i-1}, write(Y,N_1),V>, <T_i, read(X,N_1),V>.

BUPT

7.4.6 Controlul optimist al concurenței

În multe aplicații, mecanismele bazate pe *lock*-uri restring concurența și generează o regie de sistem sporită. În ([BG81][Gos91]) se arată că mecanismele bazate pe *lock*-uri prezintă următoarele dezavantaje: (1) introduc o regie de sistem sporită nenecesară pentru tranzacțiile read-only care nu afectează integritatea datelor; (2) nu există mecanisme bazate pe *lock*-uri care să ofere un grad înalt de concurență; (3) interzice eliberarea *lock*-urilor pînă la sfîrșitul tranzacției, deși nu este absolut necesar; aceasta se face în scopul evitării abortărilor în cascadă, micșorează gradul de concurență; (4)cînd se lucrează cu obiecte mari, aflate în memoria secundară, blocarea obiectelor accesate frecvent (referite ca noduri suprasolicitate) pe timpul cît se așteaptă pentru accesul la memoria secundară poate provoca o descreștere semnificativa a concurenței; (5) este posibl ca, în unele cazuri, tranzacțiile să nu se suprapună (în accesul la aceleași date); în aceste cazuri, blocările sunt nenecesare. Ele sunt utile numai în cazurile cele mai dezavantajoase, cînd tranzacțiile accesează aceleași date.

Pentru a evita aceste dezavantaje, Kung și Robinson au prezentat pentru prima oară un algoritm pentru controlul optimist al concurenței.

7.4.6.1. Concepte de baza

Ideea de bază a mecanismelor ce oferă un control optimist al concurenței este presupunerea că, în accesul la date, conflictele apar destul de rar în execuția tranzacțiilor concurente. De aceea nu se abortează sau întîrzie nici o tranzacție în timpul desfășurării ei, ca în cazul mecanismelor bazate pe *lock*-uri sau pe marca de timp; în schimb, în momentul realizării tranzacției se execută un test de validare pentru a se detecta eventualele conflicte care au apărut și care ar fi putut afecta consistența sistemului. Dacă rezultatul testului este pozitiv, atunci tranzacția va fi abortată și restartată.

Mecanismele care oferă un control optimist al concurenței sunt adecvate pentru prelucrările în care conflictele la date apar rar.

Algoritmii pentru controlul optimist al concurenței se pot grupa în două categorii. Prima categorie ([BG81], [KR81], [CG87]) cuprinde algoritmii care construiesc graful de precedență (vezi sect. 7.4.2) și îl testează pentru determinarea aciclității. Graful relației de precedență este actualizat de cîte ori se acceptă o cerere de scriere sau citire. Dupa cum s-a aratat în secțiunea 7.4.2., absența ciclurilor în graful relației de precedență este suficientă pentru ca o planificare a unui set de tranzacții să fie serializabilă. Problema crucială în sistemele distribuite este însă proiectarea unui mecanism eficient pentru menținerea grafului relației de precedență. O soluție a acestei probleme este menținerea unei copii a grafului relației de precedență în fiecare nod și actualizarea ei la fiecare operație *read* sau *write*; aceasta soluție implică însă o circulație mare de mesaje în sistem și de aceea se utilizează mai mult cea de a doua categorie de algoritmi.

A doua categorie de algoritmi ([KR81], [CO82], [CG87], [Gos91], [BK91]) execută o tranzacție în trei faze: *o fază de citire, o fază de validare și o fază de scriere*. Pe tot parcursul *fazei de citire*, tranzacția se execută local, în zona de lucru proprie. În *faza de validare*, se testează dacă, prin execuția operațiilor de actualizare pregatite de tranzacție în zona de lucru nu se violează consistența obiectelor. Dacă testul nu este validat, urmează *faza de scriere* în care are loc scrierea obiectelor actualizate din zona de lucru în sistemul de obiecte; dacă testul nu este validat, tranzacția este abortată și eventual, restartată, ca o tranzacție nouă.

Controlul optimist al concurenței prezintă următoarele avantaje: (1) execuția tranzacțiilor care nu pereclitează consistența datelor nu este întîrziată; (2) nu pot apare interblocări și restartări ciclice, dar și următoarele dezavantaje: (1) trebuie colectate informații despre tranzacții, în vederea realizării fazei de validare; (2) este posibilă restartarea infinită.

7.4.6.2. Algoritmi pentru controlul optimist al concurenței

În aceasta secțiune vor fi prezentati doi algoritmi pentru controlul optimist al concurenței în sisteme centralizate, dupa care aceștia se vor extinde la variantele pentru sisteme distribuite ([MOO87], [CG87]).

Execuția unei tranzacții se desfășoară în trei faze: *faza de citire*, *faza de validare* și *faza de scriere*. Fazele de citire și de scriere au fost explicate în secțiunea precedentă; centrul de greutate în aceasta secțiune va cadea pe faza de validare. Vor fi prezentați doi algoritmi: *algoritmul de validare serială și de validare paralela*.

În cadrul algoritmului de *validare seriala*, fiecărei tranzacții T_i i se atașează un identificator unic $TC(T_i)$care nu este neaparat o marcă de timp, ci un număr generat de un numărator global (GTC). Identificatorii se asignează tranzacțiilor ***dupa*** execuția cu succes a fazei de validare. Fie $TC_{start}(T_i)$ cel mai mare identificator de tranzacție generat în momentul inițierii tranzacției T_i și $TC_{finish}(T_i)$ cel mai mare identificator de tranzacție generat în momentul începerii fazei de validare a lui T_i. Tranzacțiile ale căror identificatori sunt: $TC_{start}(T_i)+1, TC_{start}(T_i)+2, \ldots, TC_{finish}(T_i)$ constituie setul de tranzacții a căror fază de scriere s-a executat concurent cu faza de citire a lui T_i; acest set se va desemna prin $T_{st}(T_i)$. Cu alte cuvinte, $T_{st}(T_i)$se definește astfel: $T_{st}(T_i) = \{T_k : TC(T_k) \in \{TC_{start}(T_i)+1, \ldots, TC_{finish}(T_i)\}\}$

Setul de date citite de tranzacția T_i va fi desemnat prin $O_r(T_i)$, iar cel de date scrise de tranzacția T_i va fi desemnat prin $O_w(T_i)$. Cu aceste condiții, algoritmul de validare serială poate fi descris astfel:

BUPT

ValidareSeriala(T_i) ::

$TC_{finish}(T_i) = GTC$; test = **true**

fa $T_j \in T_{sf}(T_i) \rightarrow$

if $Or(T_i) \cap O_w(T_j) \neq \varnothing$ --> test = *false fi*

af

if test $\rightarrow$ GTC = GTC+1

$TS(T_i) = GTC$

< write phase >

[] *not* test $\rightarrow$ < abort T_i >

fi

Pentru a se putea garanta serializabilitatea, precedurile ValidareSeriala() trebuie executate în regim de excludere reciproca (sunt secţiuni critica). Ca atare execuţia unei faze de validare şi de scriere a unei tranzacţii blochează parţial intregul sistem deoarece nici o altă tranzacţie nu mai poate intra în aceste faze în acest timp, în schimb, fazele de citire ale altor tranzacţii pot fi executate.

Se observă că acest algoritm impune actualizarea serială a obiectelor, în ordinea identificatorilor de tranzacţii, de aceea algoritmul se numeşte de *validare seriala*.

Tranzacţiilor read-only nu li se asociază identificatori deoarece ele nu iau parte în validarea tranzacţiilor următoare din sistem; pentru aceste tranzacţii $TC_{start}(T_i) = TC_{finish}(T_i)$.

Algoritmul de. *validare paralela* permite unei faze de validare a unei tranzacţii să se desfăşoare concurent cu fazele de scriere ale altor tranzacţii. În acest algoritm se presupune că tranzacţiei i se asociază identificatorul la sfîrşitul fazei de scriere. Ca şi în algoritmul de validare serială, la iniţierea unei tranzacţii şi la sfîrşitul fazei de citire se citeşte contorul GTC pentru a determina valorile $TC_{start}(T_i)$ şi $TC_{finish}(T_i)$. Tranzacţiile cu identificatorii $TC_{start}(T_i)+1, TC_{start}(T_i)+2, \ldots, TC_{finish}(T_i)$ constituie setul $T_{sf}(T_i)$ al tranzacţiilor care au executat fazele de scriere concurent cu faza de citire a lui T_i. Tranzacţiile continute în $T_{sf}(T_i)$ şi-au terminat faza de scriere cînd T_i îşi începe faza de scriere. În plus, sistemul pastrează informaţii despre setul de tranzacţii care au terminat faza de citire dar nu au terminat faza de scriere. Acest set desemnat prin T_{activ} conţine tranzacţiile ale căror faze de scriere şi validare au fost executate concurent cu aceleaşi faze ale lui T_i. Algoritmul de validare paralela poate fi descris astfel:

ValidareParalela(T_i) ::

$TC_{finish}(T_i) = GTC$

$T_{locActiv}(T_i) = T_{activ}$ #se realizeaza o copie locala a lui T_{activ}

$T_{activ}(T_i) = T_{activ} \cup \{T_i\}$

test = *true*

fa $T_j \in T_{sf}(T_i) \rightarrow$

if $O_r(T_i) \cap O_w(T_j) \neq \varnothing \rightarrow$ test = *false fi*

af

fa $Tj \in T_{locActiv}(T_i) \rightarrow$

if $(O_r(T_i) \cup O_w(T_i)) \cap O_w(T_j) \neq \varnothing \rightarrow$ test = *false fi*

if test $\rightarrow$ <write phase>

GTC = GTC+1

$TS(T_i) = GTC$

$T_{activ} = T_{activ} - \{T_i\}$

[] *not* test $\rightarrow T_{activ} = T_{activ} - \{T_i\}$

<abort T_i>

fi

În acest algoritm numai operaţiile asupra lui GTC şi T_{activ} se execută în excludere reciprocă.

Un dezavantaj al acestui algoritm este acela că o tranzacţie Ti poate fi abortată în mod nenecesar datorită unui conflict cu o tranzacţie invalidată $T_k \in T_{locActiv}$ care este de asemenea abortată. În exemplul următor se prezintă o asemenea situaţie. Se consideră planificarea concurentă a tranzacţiilor T_1, T_2, T_3 şi T_4 din fig 7.4.6.2. şi se presupune că tranzacţiile acţionează asupra următoarelor elemente:

$O_r(T_1)=\{O_1, O_2\}$. $O_w(T_1)=\{O_1, O_2\}$
$O_r(T_2)=\{O_3, O_4\}$. $O_w(T_2)=\{O_2, O_5\}$
$O_r(T_3)=\{O_2, O_5\}$. $O_w(T_3)=\{O_1, O_6\}$
$O_r(T_4)=\{O_4, O_6\}$, $O_w(T_4)=\{O_6\}$

Validarea tranzacţiilor se desfăşoară astfel: (1) T_1 este validată fără condiţii deoarece atît $T_{sf}(T_1)$ cît şi $T_{locActiv}$ sunt vide; (2) pentru validarea lui T_2 se ia în considerare doar $T_{sf}(T_2)=\{T_1\}$; $T_{locActiv}(T_2)$este vidă. T_2 este validată deoarece $O_r(T_2)$ şi $O_w(T_1)$ sunt disjuncte; (3) pentru validarea lui T_3 se ia în considerare $T_{sf}(T_3)=\{T_1, T_2\}$; $T_{locActiv}(T_3)$

este vida. Deoarece $O_r(T_3) \cap O_w(T1)=\{O_2\}$ şi $O_r(T_3) \cap O_w(T_2)=\{O_2, O_5\}$, tranzacţia T_3 este abortata; (4) pentru validarea lui T_4 se ia în considerare $T_{sf}(T_4)=\{T_1, T_2\}$ şi $T_{locActiv}(T_4)=\{T_3\}$.Tranzacţia T_4 este abortată deoarece :

$$(O_r(T_4) \cup O_w(T_4)) \cap O_w(T_3) = \{O_6\}$$

Totuşi, tranzacţia T_4 este abortată inutil, deoarece conflictul apare cu o tranzacţie care va fi invalidata.

Pentru extinderea algoritmilor de validare serială şi paralela la sisteme distribuite se consideră următoarele: (1) se presupune că fiecare tranzacţie primeste un identificator unic global şi acest identificator se asociază tuturor subtranzacţiilor lui T_i: T_i^{Sk},... T_i^{Sl}; (2) validarea tranzacţiilor se desfăşoară pe două nivele: local şi global. Nivelul de *validare locală* implică acceptarea fiecărei subtranzacţii T_i^{Sk}, local, în nodul S_k, conform procedurilor de validare prezentate mai sus. Nivelul de *validare globală* implică acceptarea tranzacţiei distribuite T_i, dupa acceptarea tuturor subtranzacţiilor T_i^{Sk},... T_i^{Sl} (*two-phase commit*).

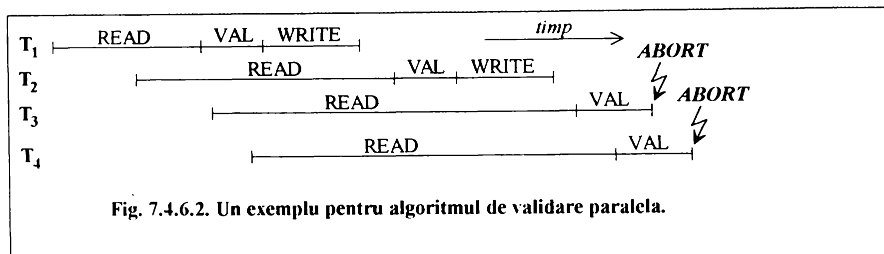


Fig. 7.4.6.2. Un exemplu pentru algoritmul de validare paralela.

7.4.7 Extinderea mecanismelor tradiţionale pentru controlul concurenţei

Mecanismele tradiţionale de control al concurenţei impun serializarea execuţiei tranzacţiilor, avînd în vedere numai operaţiile de citire şi scriere, datorită lipsei de informatii relative la operaţiile specifice aplicaţiei. Aceasta lipsă de informaţie conduce la imposibilitatea de a specifica sau verifica constrîngeri semantice asupra datelor; însă în condiţiile existentei unor astfel de informaţii, se poate construi o planificare neserializabilă dar consistentă. Prin egalizarea noţiunii de consistenţă cu noţiunea de serializabilitate se micşorează gradul de concurenţă.

7.4.7.1 Cerinţe ale controlului concurenţei în aplicaţiile avansate

Se apreciază că aplicaţiile avansate impun următoarele noi cerinţe mecanismelor de control al concurenţei([YEK87], [BK91]):

(1) Furnizarea unui suport adecvat pentru tranzacţiile lungi. În aplicaţii lungi, precum compilări de module sursă sau proiectări de circuite, operaţiile asupra obiectelor sunt operaţii de lungă durată. Dacă aceste operaţii ar fi incapsulate în tranzacţii, aceste tranzacţii vor fi, de asemenea, tranzacţii lungi, pentru care nu se accepta blocarea uneia pînă la comiterea alteia.

(2) Furnizarea unui suport care să ofere control utilizatorului. Mecanismul de control al concurenţei trebuie să furnizeze acest suport în scopul controlului de către utilizator a unor aplicaţii de natură interactivă şi nedeterministă; utilizatorul trebuie să poată iniţia o tranzacţie, executa operaţii în cadrul ei, în mod interactiv, să o poată restructura dinamic şi comite sau aborta în orice moment. Natura nedeterministă a tranzacţiilor implică faptul că mecanismul de control al concurenţei nu va putea determina dacă se violează constrîngerile de consistenţă decît în momentul execuţiei şi validării operaţiilor asupra datelor. Acest lucru poate avea ca urmare abortarea unor tranzacţii în urma unor ore de munca; de aceea utilizatorul trebuie să poată inversa explicit efectul unor operaţii, pentru a reface consistenţa datelor;

(3) Furnizarea unui suport pentru cooperare sinergetică. Cooperarea între utilizatori în timpul execuţiei unor aplicaţii precum sisteme CAD/CAM, SDE, poate avea implicaţii asupra mecanismului de control al concurenţei. În cadrul acestor medii de proiectare, este probabil ca utilizatorii să dorească schimbul de informaţii între ei la un moment dat (să partajeze anumite informaţii). Activităţile a doi sau mai mulţi utilizatori care lucrează cu aceleaşi obiecte partajate, pot să nu fie serializabile. Utilizatorii pot să-şi transmită controlul asupra obiectelor într-un mod care să poată fi realizat printr-o planificare serializabilă sau să modifice concurent două părţi ale unui obiect, cu intenţia de a le integra într-o noua versiune a obiectului. Acest mod de partajare şi schimb de informaţii a fost denumit *interacţiune sinergetică* ([YEK87], [BK91]).

În plus faţă de aceste cerinţe, multe aplicaţii avansate necesită suport pentru obiecte complexe. De exemplu, într-un sistem de proiectare software, obiectele pot fi organizate ierarhic (proiectele sunt constituite din module, modulele din funcţii). Tehnicile pentru gestiunea obiectelor complexe sunt proprii programării orientate pe obiecte; totuşi complexitatea structurii şi dimensiunea obiectelor poate avea influenţe puternice asupra mecanismului de control al concurenţei.

BUPT

7.4.7.2 Extinderea mecanismelor bazate pe serializabilitate și renunțarea la serializabilitate

Pentru rezolvarea problemelor legate de tranzacțiile lungi au fost propuse două soluții: extinderea mecanismelor bazate pe serializabilitate, care generează tot planificări serializabile și mecanisme care generează planificări neserializabile (renunțarea la serializabilitate). Ambele soluții, pentru creșterea gradului de concurență, utilizează informații semantice specifice aplicațiilor.

Extinderea mecanismelor bazate pe serializabilitate.

În mecanismele tradiționale, toate operațiile asupra obiectelor sunt abstractizate în două operații: citire și scriere. Această abstractizare este necesară pentru proiectarea unor mecanisme de control al concurenței cu caracter general, care să nu depindă de specificul aplicațiilor, de exemplu mecanismul blocării în două faze. Performanțele acestui mecanism pot fi însă inacceptabile în aplicațiile avansate, deoarece obiectele pot rămîne blocate mult timp după utilizarea lor, blocînd în acest fel alte tranzacții. Mecanismele de control optimist al concurenței oferă de asemenea performanțe slabe pentru aplicațiile avansate, deoarece cînd rata conflictelor crește, care este în general cazul tipic al aplicațiilor avansate, crește numărul de abortări și restartări ale tranzacțiilor. O rezolvare a problemelor introduse de tranzacțiile lungi este utilizarea unor informații semantice despre tranzacții și operații pentru a extinde mecanismele tradiționale. În secțiunea 7.4.8. se prezintă modalitatea de extindere a mecanismelor bazate pe lockuri și a celor de control optimist al concurenței în vederea satisfacerii cerințelor tranzacțiilor lungi.

Renuntarea la serializabilitate.

O altă soluție pentru rezolvarea cerințelor tranzacțiilor lungi este renunțarea la planificările serializabile și folosirea unor planificări care se bazează pe informațiile semantice relative fie la obiecte, fie la operațiile specifice aplicației. Mecanismele bazate pe informații semantice se împart în două mari grupuri ([SZ89]): primul grup definește și folosește proprietăți relative la concurență ale obiectelor ([SGA87]), iar cel de-al doilea utilizează informații semantice relative la tranzacții. Algoritmii din primul grup impun constrîngeri în intercalarea operațiilor tranzacțiilor pe baza conflictelor dintre operațiile, definite asupra unor obiecte de anumite tipuri. Controlul concurenței în accesul la obiecte ale căror tipuri și operații se cunosc va fi prezentat în secțiunea 7.4.9. În secțiunea 7.4.8. vor fi comentați algoritmi din al doilea grup.

7.4.8. Mecanisme pentru controlul concurenței în aplicații avansate

În această secțiune va fi prezentată extinderea mecanismelor bazate pe *lock*-uri și a celor de control optimist al concurenței la aplicații complexe.

7.4.8.1 Mecanismul blocării altruiste

Un important gen de informatii care poate fi folosit pentru a crește concurența este declararea obiectelor ca libere în momentul terminării utilizării lor; aceste obiecte vor putea fi folosite de alte tranzacții. Un mecanism care folosește aceasta tehnică este *mecanismul blocării altruiste* ([SGA87], [BK91]). Acest mecanism utilizează informații despre așa numitele *șabloane de acces* la obiecte ale unei tranzacții pentru a decide ce obiecte pot fi eliberate. Se folosesc două tipuri de informatii: *sabloane de acces negativ*, care descriu obiectele care nu vor fi accesate de tranzacție și *sabloanele de acces pozitiv* care descriu obiectele ce vor fi accesate de tranzacție și ordinea de acces. Împreună, aceste șabloane permit tranzacțiilor lungi să elibereze obiectele dupa utilizarea lor. Setul de obiecte care au fost blocate și apoi eliberate de o tranzacție lungă se numește *urma tranzacției* ("*wake*"). Eliberarea unui obiect este o operație *unlock* condiționată deoarece permite altor tranzacții să folosească obiectul numai dacă respectă cele două condiții de mai jos, care asigură serializabilitatea.

O planificare în *două faze cu eliberare* ("*two phase with release*") se definește ca o planificare care respecta condițiile:

Condiția 1

Două tranzacții nu pot deține *lock*-uri asupra aceleiasi date, simultan, cu excepția cazului cînd una dintre tranzacții a eliberat *lock*-ul înainte ca cealaltă să-l obțină. În acest caz, *lock*-ul deținut de ultima se spune că se afla în *urma* primei tranzacții.

Condiția 2

Dacă o tranzacție este în *urma* altei tranzacții atunci ea trebuie să fie complet în *urma* acelei tranzacții. Aceasta înseamnă că dacă tranzacția T_2 obține un *lock* asupra unui obiect eliberat de tranzacția T_1, atunci orice obiect accesat de T și T_2 care este deținut la momentul curent de T_2, trebuie să fi fost eliberat de T_1 înainte de a fi fost obținut de T_2.

Aceste două condiții de protocol garantează serializabilitatea tranzacțiilor fără a altera structura lor. Protocolul presupune că tranzacțiile sunt *programate* și nu *controlate* de utilizator. Exemplul următor ilustrează acest mecanism. Consideram aplicația din figura 7.4.8.1.1 în care se presupune că o echipa de proiectanți dezvoltă modular un sistem de circuite electronice. Fiecare modul din proiect conține descrierea unui anumit număr de placi (subobiecte). Presupunem că utilizatorul X, fiind un membru nou în cadrul proiectului compus din proiectantii U și V dorește să se familiarizeze cu toate schemele. În aceasta situație, X va iniția o tranzacție nouă, T_X, care va accesa toate subobiectele, pe rînd; tranzacția T_X va fi o tranzacție lungă în decursul căreia X va citi fiecare subobiect, va adăuga cîteva comentarii și va elibera subobiectul. În același timp, utilizatorul U va iniția o altă tranzacție (scurtă),

BUPT

T_U, care va accesa doar două subobiecte p_1 și p_2 din modulul A. Presupunem că T_X a accesat și eliberat deja p_2, în momentul curent fiind în citirea lui p_1. T_U trebuie să aștepte pînă cînd T_X termină utilizarea lui p_1; în acest moment T_U poate accesa p_1, gasindu-se în *urma* lui T_X. T_U poate intra în *urma* lui T_X deoarece toate obiectele la care U dorește acces (p_1 și p_2) sunt în *urma* lui T_X. Dupa terminarea lucrului cu p_1, T_U poate accesa imediat p_2, fără a întîrzia, deoarece p_2 a fost eliberat de T_X. Presupunem acum că V inițiază o altă tranzacție (scurta), T_V, în care dorește să acceseze p_2 și p_3, care nu este în *urma* lui T_X. T_V poate accesa p_2 dupa ce T_U termină tranzacția, dar, apoi trebuie să aștepte pînă fie T_X eliberează pe p_3 (cu alte cuvinte p_3 intră în urma lui T_X) sau pînă T_X se termină. Dacă T_X nu accesează niciodată p_3, T_V este fortată să aștepte terminarea lui T_X. Pentru a crește concurența în aceste de situații, în [SGA87] s-a propus un mecanism pentru extinderea dinamică a *urmei* unei tranzacții lungi, pentru a permite tranzacțiilor scurte care sunt deja în *urma* unei tranzacții lungi să-și continue execuția. Acest mecanism utilizează informații despre obiectele care *nu* vor fi accesate de o tranzacție, în scopul de a introduce aceste obiecte automat în *urma* tranzacției. Continuînd în acest mod exemplul, p_3 va fi adăugat automat în urma lui T_X, ceea ce va permite tranzacției T_V să acceseze p_3 fără întîrziere. Figura 7.4.8.1.2. prezintă acest exemplu. Fiecare tranzacție este reprezentată printr-o linie frîntă, care unește subobiectele accesate în cadrul tranzacției. Tranzacția lungă T_X accesează subobiectele p_4, p_5, p_2, p_1, p_8, p_9, p_{10}, p_6 și p_7; p_3 nu este accesat de T_X (linia groasă din figură nu aparține tranzacției lungi). Tranzacția T_U este complet în urma lui T_X, deoarece fiecare subobiect accesat de T_U (p_1 și p_2) a fost accesat înainte de T_X. În schimb, T_V nu se afla în *urma* lui T_X. Pentru a se permite execuția lui T_V, se expandează *urma* lui T_X adaugandu-se p_3 (în figura acest lucru se simbolizează cu linia groasa). Planificarea celor trei tranzacții este serializabila și echivalentă cu execuția serială T_X, T_U și T_V.

Avantajul principal al mecanismului blocării altruiste este posibilitatea de a utiliza eliberarea obiectului înainte de terminarea tranzacțiilor. Mecanismul generează planificări serializabile iar în cazul absenței informatiilor se reduce la blocarea standard în două faze. Prezintă însă un important dezavantaj: nu evită abortările în cascadă, cînd o tranzacție lungă este abortată, toate tranzacțiile scurte aflate în urma ei trebuie abortate, *chiar dacă sunt deja terminate*.

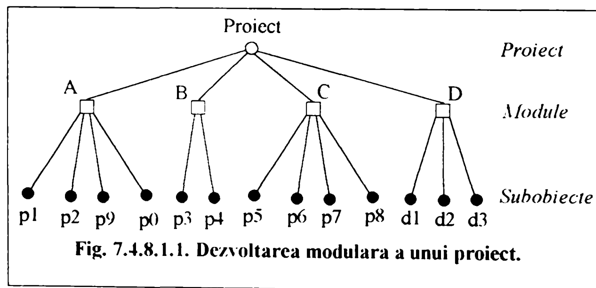


Fig. 7.4.8.1.1. Dezvoltarea modulara a unui proiect.

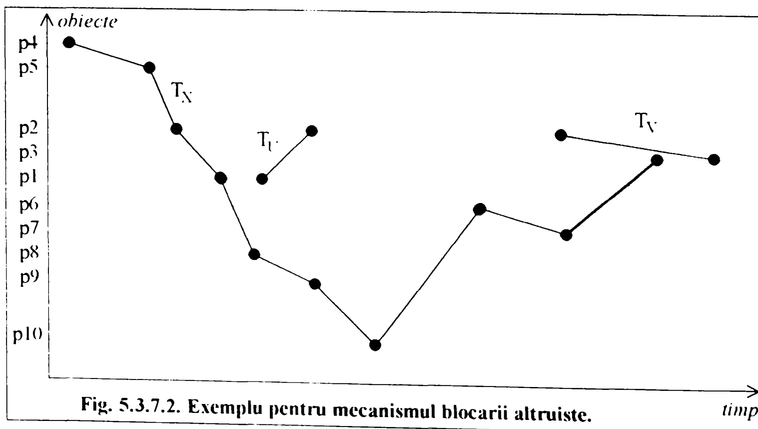


Fig. 5.3.7.2. Exemplu pentru mecanismul blocarii altruiste.

BUPT

7.4.8.2 Mecanismul validării instantanee

Mecanismul blocării altruiste are la baza mecanismul tradițional de blocare în două faze (2PL) și ca atare moşteneste unele dezavantaje ale acestuia (de exemplu, poate provoca interblocări). O alternativă este folosirea mecanismului de control optimist al concurenței, ca punct de plecare. Mecanismele tradiționale de validare a tranzacțiilor prezintă însă și ele dezavantaje precum: (1) necesitatea de a aborta o tranzacție datorită unui conflict în faza de validare, chiar dacă acest conflict nu conduce la o violare a criteriului de serializabilitate, (2) irosirea timpului datorită descoperirii conflictelor în faza de validare, (3) riscul restartărilor infinite, iar în cazul validării paralele, a restartărilor ciclice. Dezavantajul de tipul (1), și ca urmare și (2) provine din definiția prea largă a noțiunii de conflict. În [PSU86] se arată că riscul restartării inutile a tranzacțiilor poate fi micşorat dacă se împart conflictele de acces la date în două categorii: *conflicte grave* în cazul cărora tranzacțiile trebuie abortate și apoi restartate și *conflicte aparente* în cazul cărora tranzacțiile își pot continua execuția. Tot în [PSU86] se propune mecanismul validării instantanee a tranzacțiilor care utilizează distincția între cele două tipuri de conflicte. Pentru a ilustra mecanismul validării instantanee se consideră exemplul din figura 7.4.8.2. Presupunem planificarea tranzacțiilor T_X, T_U, T_V din fig.7.3.7.3. Conform mecanismului tradițional de validare, T_U și T_V ar trebui să fie abortate datorită conflictelor cu tranzacția T_X:

$O_W(T_X) \cap O_R(T_U)=\{p1\}$

$O_W(T_X) \cap O_R(T_V)=\{p2\}$

Cu toate acestea, conflictul dintre T_U și T_X este diferit față de conflictul dintre T_V și T_X. Tranzacția T_U citește subobiectul p_1, care este apoi modificat de T_X; astfel $T_U \rightarrow_{p1} T_X$. Deoarece în faza de scriere, T_U poate accesa iarăsi pe p_1, se poate spune că și $T_X \rightarrow_{p1} T_U$ și deci se formează un ciclu în graful relației de precedență. Astfel, execuția concurentă (din figură) a tranzacțiilor T_X și T_U violează criteriul de serializabilitate și de aceea T_U trebuie abortată. Conflictul dintre T_V și T_X, dimpotrivă, nu violează criteriul de serializabilitate. Tranzacția T_V citește subobiectul p_2 dupa ce acesta a fost actualizat de T_X; astfel, $T_X \rightarrow_{p2} T_V$. Deoarece algoritmul de validare actualizează subobiectele în conformitate cu ordinea identificatorilor de tranzacții, pentru orice subobiect p, $p \in \{O_W(T_X) \cap O_W(T_V)\}$ relația $T_X \rightarrow_p T_V$ este adevarată. Astfel, nu este necesară abortarea tranzacției T_V.

O altă constatare care se poate face în urma analizei acestui exemplu este aceea că acest conflict dintre T_X și T_U poate fi detectat mai devreme decît în faza de validare a lui T_U: în momentul în care T_X se termină (momentul snapshot din figura). Detectarea mai timpurie a conflictelor poate mări eficacitatea unui algoritm de validare.

Prima modificare a mecanismului tradițional de validare a tranzacțiilor constă în adăugarea în setul $O_r(T_i)$ al unei tranzacții T_i a identificatorilor tranzacțiilor care și-au terminat faza de scriere în timpul fazei de citire a lui T_i (identificatori *end-of-transaction*); acesti identificatori vor fi desemnati prin EOT_i. Setul $O_r(T_i)$ poate fi ordonat în ordinea cronologică a execuției operațiilor de citire. De aceea, pentru a valida tranzacția T_i în raport cu o tranzacție $T_j \in T_{SF}(T_i)$este suficient să se considere numai un subset al lui $O_r(T_i)$ de la începutul fazei de citire a lui T_i pînă la aparitia lui EOT_j. Fie, de exemplu setul $O_r(T)$ al unei tranzacții T ordonat astfel: $O_r(T)=\{a, b, EOT_i, c, EOT_j, d, e, EOT_k\}$. Potrivit mecanismului tradițional de validare a tranzacției T în raport cu $T_{SF}(T)=\{T_i,T_j,T_k\}$ sunt necesare următoarele teste:

$O_W(T_i) \cap \{a,b,c,d,e\}$

$O_W(T_j) \cap \{a,b,c,d,e\}$

$O_W(T_k) \cap \{a,b,c,d,e\}$

Prin modificarea mecanismului de validare, sunt necesare testele:

$O_W(T_i) \cap \{a,b\}$

$O_W(T_j) \cap \{a,b,c\}$

$O_W(T_k) \cap \{a,b,c,d,e\}$

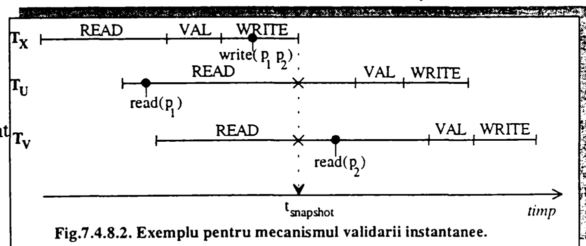


Fig.7.4.8.2. Exemplu pentru mecanismul validarii instantanee.

A doua modificare a algoritmului constă în substituirea procedurii de validare (serială sau paralelă), executată o singură dată pe durata de viață a unei tranzacții printr-un *număr de validări instantanee*. Validările instantanee se execută la sfîrțitul fazei de scriere a fiecărei tranzacții. Fie T_i o tranzacție în momentul terminării fazei de scriere; în acest moment, fiecare tranzacție T_j aflată în faza de citire trebuie validată în raport cu T_i. Setul $O_r(T_j)$ conținînd datele citite de T_j pînă la sfîrșitul fazei de scriere a lui T_i se compară cu $O_W(T_i)$. Dacă intersecția lor este nevidă, atunci T_j trebuie abortată deoarece a citit date care au fost actualizate de T_i. Fazele de scriere și validare se execută în regim de excludere reciprocă.

În concluzie, mecanismul de validare instantanee, față de mecanismul tradițional de validare are următoarele avantaje: (1) numărul de tranzacții care se abortează este mai mic; (2) conflictele de acces la date sunt detectate mai devreme, prevenind irosirea unui timp suplimentar, pînă la validarea tranzacției; (3) setul de date citite de o tranzacție care se folosesc pentru validare este mai mic și astfel detectarea conflictelor se face mai rapid.

BUPT

7.4.9. Mecanisme pentru controlul concurenţei bazate pe informaţii semantice. Sincronizarea accesului la tipuri de date abstracte

Operaţiile care se execută asupra tipurilor abstracte partajate trebuie să respecte următoarele cerinţe [Gos91][CG87][SP84]:(1). Între apelul ultimei operaţii şi execuţia ei, obiectul poate să nu fie într-o stare consistentă. Nu se permite însă altor operaţii să observe sau să manipuleze obiectul în acest interval; (2). Cînd un grup de tranzacţii apelează operaţii ce se execută asupra unui obiect partajat, aceste operaţii trebuie astfel intercalate încît să menţină seriabilitatea tranzacţiilor sau să respecte o *proprietate de ordine* (mai puţin restrictivă decît seriabilitatea) în grupul tranzacţiilor; (3). Operaţiile efectuate asupra obiectelor partajate trebuie să nu poată observa informaţii care se pot modifica dacă o tranzacţie va fi invalidată. Acest lucru necesită întîrzierea execuţiei unor operaţii aparţinînd unor tranzacţii, pîna ce o altă tranzacţie este terminată, cu succes sau fără succes.

Metodele de sincronizare a accesului la tipuri abstracte partajate prezentate în acest capitol se concentrează asupra respectării cerinţelor 2 şi 3 de mai sus, tipice sistemelor bazate pe tranzacţii şi reliefează importanţa informaţiilor semantice în extinderea concurenţei.

7.4.9.1. Relaţia de dependenţă

S-a văzut că intercalarea operaţiilor unui grup de tranzacţii este descrisă de planificări. O ***planificare*** poate fi definită ca o secvenţă de perechi <tranzacţie, operaţie> care semnifică ordinea în care sînt intercalate operaţiile componente unor tranzacţii (vezi 7.4.2.).

Pentru a analiza tranzacţii care conţin operaţii asupra unor tipuri abstracte arbitrare, planificatoarele trebuie să aibă caracterizate explicit operaţiile. De exemplu, o planificare poate conţine operaţii pentru a introduce un element într-o coadă, pentru a insera o intrare într-un director, pentru a insera un nod într-un arbore B. Deoarece tipurile abstracte pot fi implementate într-o manieră arbitrară sînt necesare doua tipuri distincte de planificări: ([SP84]): (1)***planificările abstracte*** caracterizează proprietăţile de consistenţă a unui tip abstract, indiferent de implementare; (2)***planificările de apel*** caracterizează proprietăţile de concurenţă ale unei implementări.

Operaţiile apelate într-o planificare abstractă reprezintă operaţii furnizate de o implementare ideală a tipului abstract. Această implementare este ideală în sensul că nu este permis să se modifice ordinea operaţiilor executate asupra unui obiect. Astfel, planificările abstracte conţin operaţiile în ordinea în care ele afectează obiectele partajate. De exemplu, să considerăm tipul abstract *listă FIFO* (o coadă) şi operaţiile de introducere a unui element la sfîrşitul cozii, QEnq, respectiv de extragere a unui element de la începutul cozii, QDeq. Presupunem că iniţial coada Q este vidă. Atunci, planificarea abstractă:

t1: <T_1, QEnq(Q,S),X>
t2: <T_2, QEnq(Q,S),Y>
t3: <T_3, QDeq(Q,S),V>

descrie exact starea cozii la fiecare moment t1, t2, t3. Astfel, se poate deduce că a 3-a operaţie extrage în variabila V elementul X, iar în coadă rămîne numai elementul Y. Proprietăţile de consistenţă ale unui tip pot fi specificate sub forma unui set de planificări abstracte. Acestea sînt denumite ***planificări consistente*** şi o descriere a acestui set este o parte a specificării tipului.

Implementările reale pot reordona operaţiile executate asupra unui obiect pentru a mări concurenţa, fără a modifica proprietăţile de consistenţă. Să considerăm, de exemplu, o implementare a unui tip coadă în care elementele destinate a fi introduse în coadă sînt mai întîi depozitate într-un buffer local al tranzacţiei şi introduse în bloc la sfîrşitul tranzacţiei. Această implementare permite oricărui număr de tranzacţii să apeleze QEnq simultan, cu restricţia serializării corecte cînd sînt implementate mai multe cozi. Prin execuţia operaţiilor de introducere în bloc, această implementare reordonează efectiv operaţiile QEnq. Planificările de apel intercalează operaţiile în ordinea apelurilor implementărilor reale care poate fi alta decît cea a implementării abstracte. De exemplu, utilizînd implementarea reală de mai sus a unui tip coadă, poate rezulta următoarea planificare de apel:

<T_2, QEnq(Q,S), Y>
<T_1, QEnq(Q,S), X>
<T_3, QDeq(Q,S), V>

Dacă T_1 se termină normal înaintea lui T_2 implementarea reală a reordonat cele două operaţii de introducere.

Corespondenţa între planificările de apel şi planificările abstracte este de tipul mai multe la unul; fiecare planificare de apel implementează exact o planificare abstractă, dar o planificare abstractă poate fi implementată prin mai multe planificatoare de apel. Strategia de sincronizare utilizată de o implementare determină un set de planificări de apel denumite ***planificări legale***, care sînt permise de implementare.

Prin examinarea unui planificator abstract, este posibil să se determine ce dependenţe există între tranzacţiile

BUPT

implicate. Notația $D{:}T_i{:}X \rightarrow_o T_j{:}Y$ va fi utilizată pentru a reprezenta dependența D formată cînd tranzacția T_i execută operația X și ulterior, tranzacția T_j execută operația Y asupra aceluiași obiect O. Setul de perechi ordonate $\{(T_i, T_j)\}$ pentru care există X, Y și O astfel încît $D{:}T_i{:}X \rightarrow_o T_j{:}Y$ formează o relație, notată $<_D$. Dacă $T_i <_D T_j$, se spune că T_i precede pe T_j și T_j depinde de T_i, prin dependența D.

Dacă se presupune că toate operațiile asupra obiectelor sînt alese arbitrar si nu se iau în considerare informații semantice utile pentru implementarea mecanismului de concurență, singura cerință este ca fiecare tranzacție, privită individual, să transforme obiectul dintr-o stare inițială consistentă într-o stare finală consistentă. În aceste condiții, numai planificatoarele abstracte serializabile pot garanta că păstrează corectitudinea tranzacțiilor individuale. Urmînd presupunerea că toate operațiile sînt nedistingibile, numai o singură posibila dependență D poate fi definită astfel: $T_1 <_D T_2$, dacă T_1 execută orice operație asupra unui obiect, înainte ca T_2 să execute orice operație asupra acestui.

Definiție.

Fie $<_D^*$ închiderea tranzitivă a relației $<_D$. O planificare este ***ordonabilă*** referitor la {D} dacă $<_D^*$ este o relație de ordine. Cu alte cuvinte, nu există cicluri de forma

$$T_1 <_D T_2 <_D \dots <_D T_n <_D T_1.$$

În general, o planificare este ordonabilă, referitor la S, unde S este un set de relații de dependență, dacă toate relațiile din S au o închidere tranzitivă de ordine parțială. Relațiile din S sînt denumite ***relații prescrise***.

Se poate demonstra că ordonabilitatea referitoare la $\{<_D\}$ este echivalentă cu seriabilitatea.

Luînd în considerare informații semantice despre operațiile aplicate asupra obiectelor în tranzacții, este posibil să se identifice relații de dependență pentru care este permisă formarea de cicluri. Să considerăm, de exemplu, mecanismul tradițional *read/write* de control al concurenței în accesul la un obiect. Un asemenea sistem implementează doua tipuri de operații asupra obiectelor: *read_object*(R) și *write_object*(W). Există relații de dependență posibile între două tranzacții care accesează același obiect:

1. $D_1{:}T_i{:}R \rightarrow_o T_j{:}R$; T_j citește un obiect, ulterior citit de T_j
2. $D_2{:}T_i{:}R \rightarrow_o T_j{:}W$; T_j citește un obiect, ulterior modificat de T_j
3. $D_3{:}T_i{:}W \rightarrow_o T_j{:}R$; T_i modifică un obiect, ulterior citit de T_j
4. $D_4{:}T_i{:}W \rightarrow_o T_j{:}W$; T_i modifică un obiect, ulterior modificat de T_j

Mecanismul anterior, nedistingînd aceste dependințe, previne formarea de cicluri în relația de dependență $<_D$, reuniunea tuturor patru de mai sus. În contrast, mecanismul de control al concurenței Read/Write ține însă cont de faptul că dependințele $R \rightarrow R$ nu influențează comportarea sistemului. Astfel, fiind dată o pereche de tranzacții T_1 și T_2, o planificare abstractă în care atît T_1 cît și T_2 execută o operație Read asupra unui obiect partajat, semantica operațiilor Read asigură faptul că nici T_1, nici T_2 și nici altă tranzacție din planificare nu determină dacă $T_1 <_{D1} T_2$ sau $T_2 <_{D1} T_1$.

În cazul *read/write*, condiția necesară de seriabilitate poate fi reformulată, în termenii relațiilor de dependență astfel: o planificare este serializabilă dacă este ordonabilă referitor la $\{_{D2 \cup D3 \cup D4}\}$. Permițînd operații de citire multiple, mecanismul *read/write*, permite formarea de cicluri în dependența $<_{D1}$ și în relații compuse care includ $<_{D1}$ prevenind însă formarea de cicluri în relații compuse din reuniuni $<_{D2}$, $<_{D3}$ și $<_{D4}$. De exemplu, să considerăm următoarele planificări, cu efecte identice în starea sistemului:

P1:: $<T_1, R(O,S),A>$
$<T_2, R(O,S),B>$
$<T_1, W(O,S),C>$

P2:: $<T_2, R(O,S),B)>$
$<T_1, R(O,S),A)>$
$<T_1, W(O,S),C)>$

În prima planificare, $T_1 <_{D1} T_2$ și $T_2 <_{D2} T_1$. Există deci un ciclu în relația $<_{D1 \cup D2}$, dar nu există cicluri în relațiile $<_{D2 \cup D3 \cup D4}$. În a doua planificare, primele două operații sînt inversate și nu se formează astfel nici un ciclu.

Pe de altă parte, următoarele două planificări nu au efecte identice:

P1:: $<T_1, R(O,S),A>$
$<T_2, W(O,S),B>$
$<T_1, W(O,S),C>$

P2:: $<T_2, W(O,S),B>$
$<T_1, R(O,S),A>$
$<T_1, W(O,S),C>$

BUPT

În acest caz, prima planificare nu este serializabilă deoarece $T_1 <_{D2} T_2$ şi $T_2 <_{D4} T_1$, deci se formează astfel un ciclu în relaţia $<_{D2 \cup D4}$. Într-adevăr, T_1 observă O înainte de a fi modificat de T_2, dar starea finală a lui O reflectă modificarea W din T_1 şi nu din T_2, implicînd o execuţie a lui T_1 înainte de T_2. A doua planificare nu are cicluri şi deci este seriabilă.

În concluzie, se poate afirma că ***ordonabilitatea cu referire la un set de relaţii de dependenţă prescrise furnizează o modalitate de caracterizare a planificatoarelor consistente***. Pentru un control al concurenţei care forţează seriabilitatea fără informaţii semantice, setul de relaţii prescrise conţine $<_D$, care este echivalentă cu reuniunea tuturor relaţiilor de dependenţă posibile. Pentru un mecanism Read/Write, setul conţine relaţia compusă $<_{R \rightarrow W \cup W \rightarrow R \cup W \rightarrow W}$. Cînd se iau în considerare informaţii specifice tipului, relaţiile prescrise sînt de asemenea specifice tipului abstract. În unele cazuri aceste proprietăţi sînt echivalente cu seriabilitatea, dar, în general, un grad mai mare de concurenţă se obţine numai cînd sînt garantate proprietăţi mai puţin restrictive.

În afară de utilitatea lor în ceea ce priveşte definirea consistenţei, relaţiile de dependenţă sînt utile şi în detectarea situaţiilor ***de abort în cascadă***. Un abort în cascadă este posibil atunci cînd se formează o dependenţă între două tranzacţii, prima fiind neterminată. Un abort al primei tranzacţii poate genera un abort în cascadă a celor ce depind de ea. Condiţiile în care un abort în cascadă apare efectiv depind de relaţia de dependenţă implicată şi de proprietăţile acelui obiect. Dacă considerăm cele patru relaţii de dependenţă prezentate în mecanismul de control al concurenţei Read/Write atunci: (1) relaţia de dependenţă $R \rightarrow R$ nu va cauza niciodată abort în cascadă; (2) analog, relaţiile $R \rightarrow W$ şi $W \rightarrow W$ nu pot genera abort în cauză, deoarece, în ambele cazuri, a doua tranzacţie nu depinde de datele modificate de prima tranzacţie; (3) În absenţa unor informaţii semantice suplimentare, pentru relaţia $W \rightarrow R$ se poate presupune că un abort al primei tranzacţii o va afecta pe a doua, care trebuie şi ea terminată prin abort.

După detectarea dependenţelor care pot genera abort în cascadă, trebuie controlată formarea lor. Transpusă în termenii planificărilor abstracte, afirmaţia devine:

Teoremă.

Considerînd operaţiile care formează dependenţa, acestea nu trebuie să se intercaleze în două tranzacţii, dacă prima tranzacţie nu este terminată înaintea celei de-a doua.

Trebuie făcută observaţia că nu toate planificatoarele consistente sînt şi fară abort în cascadă şi invers. Mecanismele de sincronizare trebuie să restrîngă setul de planificări de apel legale la planificările abstracte aflate la intersecţia celor consistente şi celor fără cascadă.

7.4.9.2. Specificarea tipurilor abstracte

Metodele tradiţionale pentru sincronizarea accesului la o instanţiere a unui tip de date abstract partajat sînt destinate să asigure corectitudinea individuală a operaţiilor asupra unui obiect, cu alte cuvinte să prevină anomaliile datorate concurenţei. Într-un sistem bazat pe tranzacţii, suplimentar, pe lîngă corectitudinea individuală a operaţiilor, trebuie asigurată pastrarea proprietăţilor de consistenţă la nivelul întregii tranzacţii, care poate conţine un număr arbitrar de operaţii.

Acest subcapitol, prin compararea unui tip listă serializabilă cu un tip semilistă care restricţionează cerinţele de consistenţă, arată că dacă se exploatează la maximum informaţiile semantice pentru operaţii, gradul de concurenţă poate fi crescut prin renunţarea la seriabilitate şi înlocuirea ei cu ordonabilitatea referitoare la un set de relaţii de dependenţă.

A. Directoare

Ca prim exemplu, se consideră tipul de date *director*, creat în scopul menţinerii unei corespondenţe între un şir alfanumeric (cheie), şi pointeri pentru obiecte arbitrare. Operaţiile uzuale furnizate sînt:

1. *InsertDir(dir,str,ptr)*: se inserează obiectul *ptr* cu cheia *str* în directorul *dir*. Se returnează OK sau cheie duplicată.
2. *DeleteDir(dir,str)*: se şterge obiectul cu cheia *str* din catalagul *dir*. Se returnează OK sau negăsit.
3. *LookupDir(dir,str)*: se caută cheia *str* în *dir*. Se returnează *ptr* sau negăsit.
4. *Dumpdir(dir)*: se returnează un vector de perechi <str,ptr> cu conţinutul complet al directorului *dir*.

Intercalarea acestor operaţii apelate de tranzacţii executate concurent trebuie controlată cu mare grijă. Tipul director trebuie să aibă următoarele proprietăţi:

1. Dacă o tranzacţie ia în considerare o intrare cu cheia "S" dintr-un director, toate operaţiile următoare ale aceleiaşi tranzacţii trebuie să poată accesa această intrare cu cheia "S" (numai dacă, evident, una dintre aceste operaţii nu o şterge). Nici unei alte tranzacţii nu îi va fi permis, însă, să şteargă intrarea "S" sau să modifice pointerul care îi corespunde înainte ca tranzacţia care o accesează să se termine.

2. Dacă o tranzacţie constată că un director ***nu*** conţine o intrare cu cheia "S", nici unei alte tranzacţii nu i se poate permite să creeze o intrare cu această cheie, pînă la terminarea primei tranzacţii.

3. Dacă o tranzacţie crează o intrare nouă într-un director, nici unei alte tranzacţii nu i se poate permite să observe

BUPT

această intrare pînă cînd prima tranzacție nu se termină cu succes.

4. Dacă o tranzacție șterge o intrare dintr-un director D și o inserează într-un alt director D', nu trebuie să fie posibil ca o altă tranzacție să observe intrarea respectivă fie în D fie în D'.

O modalitate de respecta aceste proprietăți este prin modelarea operațiilor *InsertDir* și *DeleteDir* ca o operație de citire urmată de o operație de scriere și a operațiilor *Lookup* și *DumpDir* ca o operație de citire. Tipul director poate atunci fi specificat utilizînd relațiile de dependență *read/write* discutate în subcapitolul anterior.

Aceste informații semantice sînt însă insuficiente și restrîng concurența în mod nenecesar. Să presupunem, de exemplu, că implementăm tipul director utilizînd mecanismul standard de blocare *read/write* în două faze. Considerînd operația *LookupDir(dir,"PSDR")* care, încercînd să obțină un *lock* S, ea va fi blocată de o altă tranzacție care a executat *DeleteDir(dir,"PNL")* și care deține un *lock* X asupra obiectului director. Însă operația *LookupDir(dir,"PSDR")* nu depinde în nici un caz de execuția operațieie *DeleteDir(dir,"PNL")* care poate fi mai tîrziu și abortată, astfel încît blocarea tranzacției nu este necesară. De aceea, tranzacția care deține *lock*-ul X, poate fi, din acest punct de vedere o tranzacție oricît de lungă. Această situație este un exemplu în care gradul de concurența este redus nemotivat; cauza este lipsa de informații semantice despre tipul director. Prin utilizarea de informații semantice, blocajul nemotivat poate fi înlăturat. Următorul set de regului de sincronizare care utilizează informații mai detaliate despre operațiile relative la un director pot fi folosite pentru a specifica modul în care multiple tranzacții pot fi intercalate fără a viola seriabilitatea și fără a permite abort în cascadă:

1. Operațiile *InsertDir* sau *DeleteDir* care specifică chei diferite pot fi executate în orice ordine.
2. Operațiile *InsertDir*, *DeleteDir*, sau *LookupDir* care specifică aceeași cheie ca alte operații *InsertDir* sau *DeleteDir* executate anterior în tranzacții neterminate, trebuie blocate pînă la terminarea acelor tranzacții.
3. Operațiile *LookupDir* pot fi executate în orice ordine.
4. Operațiile *InsertDir* care specifică aceeași cheie ca și operații anterioare *LookupDir* nereușite executate de alte tranzacții neterminate trebuie blocate pînă acele tranzacții sînt terminate.
5. Operațiile *DeleteDir* care specifică aceeași cheie ca și operații anterioare *LookupDir* reușite executate de alte tranzacții neterminate trebuie blocate pînă acele tranzacții sînt terminate.
6. Operațiile *DumpDir* trebuie blocate dacă au fost executate operații *InsertDir* sau *DeleteDir* de alte tranzacții neterminate.
7. Operațiile *InsertDir* sau *DeleteDir* trebuie blocate dacă operații DumpDir au fost executate în alte tranzacții neterminate.

Aceste specificații pot fi precizate utilizînd relații de dependență și ordonabilitatea. Prima etapă este identificarea dependențelor pentru acest tip. Ca și regulile de sincronizare ele se bazează pe operațiile definite pentru acel tip. Pentru a păstra numărul de dependențe la minimum operațiile pentru tipul director pot fi împărțite în trei grupuri:

1. Acele operații care modifică o intrare particulară într-un director, identificat de cheia α. Operațiile *InsertDir* și *DeleteDir* terminate cu succes, sînt în această clasă. Ele sînt denumite operații *Modify* (M).
2. Acele operații care observă prezența, absența sau conținutul unei intrări particulare într-un director, identificată printr-o cheie α. Operațiile *LookupDir* ca și operațiile *InsertDir* și *DeleteDir* nereușite sînt în această clasă. Această clasă este clasa operațiilor *Lookup* (L).
3. Acele operații care observă proprietăți ale directorului care nu pot fi izolate într-o intrare individualizată. Singura operație din această clasa este *DumpDir* (dintre cele definite; mai pot fi date și alte exemple, astfel poate fi definita o operație care returnează numărul de intrări din director). Această clasă este clasa operațiilor *Dump* (D).

Suplimentar definirii operațiilor ca atare, regulile de sincronizare iau în considerare și argumentele furnizate de operații. În următorul set complet de dependențe, simbolurile α și α' reprezintă chei distincte în operații cu directoare:

1. $D_1{:}T_i{:}M(\alpha) \rightarrow T_j{:}M(\alpha')$; T_i modifică o intrare cu cheia α și T_j modifică ulterior o intrare cu o cheie diferită α'.
2. $D_2{:}T_i{:}M(\alpha) \rightarrow T_j{:}M(\alpha)$; T_i modifică o intrare cu cheia α și T_j modifică ulterior aceeași intrare.
3. $D_3{:}T_i{:}M(\alpha) \rightarrow T_j{:}L(\alpha')$; T_i modifică o intrare cu cheia α și T_j observă ulterior o intrare cu o cheie diferită α'
4. $D_4{:}T_i{:}M(\alpha) \rightarrow T_j{:}L(\alpha)$; T_i modifică o intrare cu cheia α și T_j observă ulterior aceeași intrare.
5. $D_5{:}T_i{:}L(\alpha) \rightarrow T_j{:}L(\alpha')$; T_i observă o intrare cu cheia α și T_j observă ulterior o intrare cu o cheie diferită α'.
6. $D_6{:}T_i{:}L(\alpha) \rightarrow T_j{:}L(\alpha)$; T_i observă o intrare cu cheia α și T_j observă ulterior aceeași intrare cu cheia α.
7. $D_7{:}T_i{:}L(\alpha) \rightarrow T_j{:}M(\alpha')$; T_i observă o intrare cu cheia α și T_j modifică ulterior o altă intrare cu cheia α'.
8. $D_8{:}T_i{:}L(\alpha) \rightarrow T_j{:}M(\alpha)$; T_i observă o intrare cu cheia α și T_j modifică ulterior aceeași intrare.
9. $D_9{:}T_i{:}D \rightarrow T_j{:}M(\alpha)$; T_i observă întregul conținut al directorului și T_j modifică ulterior o intrare cu cheia α.
10. $D_{10}{:}T_i{:}D \rightarrow T_j{:}L(\alpha)$; T_i observă întregul conținut al directorului și T_j observă o intrare cu cheia α.
11. $D_{11}{:}T_i{:}M(\alpha) \rightarrow T_j{:}D$; T_i modifică o intrare cu cheia α și T_j observă ulterior întregul conținut al directorului.
12. $D_{12}{:}T_i{:}L(\alpha) \rightarrow T_j{:}D$; T_i observă o intrare cu cheia α și T_j observă ulterior întregul conținut al directorului.
13. $D_{13}{:}T_i{:}D \rightarrow T_j{:}D$; T_i observă întregul conținut al directorului și T_j ulterior observă din nou întregul conținut al directorului.

Ca și dependența R → R, multe din aceste dependențe nu trebuie să afecteze tranzacțiile ulterioare. De aceea, ele nu trebuie luate în considerare la definirea condițiilor necesare pentru consistență. Dependențele următoare sînt cele care nu

BUPT

trebuie luate în considerare:

1. Acelea pentru care nici o operaţie din dependenţă nu modifică obiectul (directorul): D_6, D_{10}, D_{12}, D_{13}. Acestea sînt analoage cu dependenţa $R \rightarrow R$.

2. Acelea pentru care cele două operaţii din dependenţă referă două chei diferite: D_1, D_3, D_5, D_7.

Rezultă, în concluzie, că un abstract pentru tipul director este consistent dacă este ordonabil referitor la $\{<_{D2 \cup D4 \cup D8 \cup D9 \cup D11}\}$.

B. Liste FIFO

Specificaţii similare cu cele prezentate mai sus există şi pentru alte tipuri de date. Listele FIFO sînt un exemplu mai des întîlnit în literatură ([AC93][CG87][SP84][Mus94d]).

Luăm în considerare două operaţii asupra listei FIFO:

1. *QEnq(queue,ptr)*: adaugă o intrare la sfîrşitul cozii.

2. *QDeq(que)*: citeşte şi elimină o intrare de la începutul cozii. Dacă lista este vidă, operaţia este blocată pînă cînd lista devine nevidă.

În scopul seriabilizării tranzacţiilor care conţin operaţii asupra listelor FIFO, numeroase proprietăţi trebuie garantate. De exemplu:

1. Dacă o tranzacţie adaugă cîteva intrări într-o listă FIFO, aceste intrări trebuie să apară împreună în aceeaşi ordine la sfîrşitul listei.

2. Orice intrări adăugate într-o listă coadă în timpul unei tranzacţii nu pot fi observate de alte tranzacţii decît după ce prima tranzacţie se termină cu succes.

3. Dacă două tranzacţii crează fiecare intrări în două liste, ordinea relativă a intrărilor create de cele două tranzacţii trebuie să se regăsească şi în cele două liste.

Aceste proprietăţi, prea restrictive în ceea ce priveşte concurenţa, pot fi modificate pentru a permite execuţia concurentă a mai multor operaţii. De exemplu, dacă operaţiile *QEnq* din mai multe tranzacţii sînt intercalate, atunci intrările create de fiecare tranzacţie nu vor apare în bloc la începutul listei FIFO.

Analog dacă se permite execuţia unei operaţii *QDeq* într-o tranzacţie urmată de o altă tranzacţie neterminată care execută deasemenea o operaţie *QDeq*, un abort al unei tranzacţii va interschimba poziţiile celor două intrări.

Setul de planificatoare abstracte consistente pentru liste FIFO serializabile poate fi specificat în termeni de dependenţă şi ordonabilitate. La fel ca şi în cazul directoarelor, este necesar să se distingă individual elemente în lista FIFO. Se presupune că, în momentul execuţiei operaţiei *QEnq* fiecare element are asociat un identificator unic. În continuare, simbolurile α şi α' se vor utiliza pentru a simboliza identificatori distincţi pentru elemente diferite, iar operaţiile *QEnq* şi *QDeq* vor fi prescurtate cu E respectiv D. Setul complet de dependenţe pentru liste FIFO este:

1. D_1:T_i:E(α) $\rightarrow_Q$ T_j:E(α'); T_j introducere un element α' în coada Q după ce T_i a introdus anterior elementul α.
2. D_2:T_i:E(α) $\rightarrow_Q$ T_j:D(α'); T_j elimină elementul α' după ce T_i a introdus elementul α.
3. D_3:T_i:E(α) $\rightarrow_Q$ T_j:D(α); T_j elimină elementul α care a fost introdus anterior de T_i.
4. D_4:T_i:D(α) $\rightarrow_Q$ T_j:E(α'); T_j introduce elementul α' după ce T_i a eliminat elementul α.
5. D_5:T_i:D(α) $\rightarrow_Q$ T_j:D(α'); T_j elimină elementul α' după ce T_i a eliminat elementul α.

Folosind schema de sincronizare *read/write*, QEnq poate fi modelată ca o operaţie de scriere în timp ce QDeq poate fi modelată ca o operaţie de citire urmată de o operaţie de scriere. Atunci, conform schemei de sincronizare *read/write*, trebuie prevenită formarea de cicluri cuprinzînd relaţia de dependenţă $<_{R \rightarrow W \cup W \rightarrow R \cup W \rightarrow W}$. Totuşi, această soluţie este restrictivă într-un grad mai mare decît necesar. Considerăm, de exemplu dependenţa D_2, care se formează atunci cînd o tranzacţie elimină un element după ce o altă tranzacţie a introdus anterior un alt element sau cazul D_4, care este de fapt cazul invers lui D_2. Concurenţa poate fi mărită dacă nu se includ în mecanismul de control aceste dependenţe. Prin urmare, pentru crearea unei liste FIFO stricte trebuie garantată ordonabilitatea cu referire la relaţia $<_{D1 \cup D3 \cup D5}$ a planificărilor abstracte, dar pot fi admise cicluri cu dependenţe din relaţia D_2 sau D_4. De exemplu, considerăm următoarea planificare, în care operează două tranzacţii într-o listă FIFO care conţine iniţial elementele {A,B}:

t1: <T_1, QEnq(Q,S), X>
t2: <T_2, QDeq(Q,S), A> returnează X
t3: <T_1, QEnq(Q,S), Y>

La pasul t2 al planificării, se formează o dependenţă T1 $<_{D2}$ T2, iar la pasul t3, dependenţa T2 $<_{D4}$ T1. Astfel se formează un ciclu în relaţia compusă $<_{D2 \cup D4}$, dar planificarea este ordonabilă în sensul de mai sus.

Relaţiile de dependenţă pot fi folosite pentru a planificări şi din punct de vedere al situaţiei de abort în cascadă. Relaţia de dependenţă $<_{D1}$ este similară dependenţei $W \rightarrow W$ şi deci nu poate provoca abort în cascadă. Relaţiile $<_{D3}$ şi $<_{D5}$

BUPT

sînt similare dependenţelor W → R. Într-o dependenţă D3, informaţia este transferată în listă (sub forma unui element α) între tranzacţii; este evident că o astfel de dependenţă poate provoca abort în cascadă. O dependenţa D5 poate, similar, să cauzeze abort în cascadă deoarece eliminarea unui element de către prima tranzacţie afectează ce element este primit de a doua tranzacţie.

Deşi această definiţie pentru consistenţa unei liste FIFO este o schemă *read/write* îmbunătăţiţă, totuşi restrînge în mod nenecesar concurenţa. Ea permite la cel mult două tranzacţii şa acceseze concurent o listă FIFO, una executînd operaţii *QEnq*, iar cealaltă operaţii *QDeq*. Spre deosebire de director, o listă FIFO trebuie să păstreze şi ordinea elementelor pe care le conţine. Un sistem bazat pe tranzacţii serializabile garantează faptul că tranzacţiile pot fi plasate în orice ordine; prin forţarea unei anumite ordini a elementelor, tipuri de date precum liste FIFO sau LIFO restricţionează concurenţa.

C. Semiliste FIFO

Este posibil să se restrîngă cerinţele de integritate pe care programatorul le poate impune asupra unui tip, garantîndu-se consistenţa prin proprietăţi de consistenţă mai puţin ferme decît seriabilitatea. Acest lucru permite mărirea gradului de concurenţă şi este demonstrat în continuare printr-un tip special de listă FIFO.

Unul din modurile obişnuite de lucru cu o listă FIFO este folosirea ei ca buffer între un producător şi un consumator. În mod uzual, ordinea exactă a intrărilor din listă nu este importantă. Ceea ce este esenţial este ca intrările introduse la un capăt al cozii să ajungă garantat în celălalt capăt într-un timp aproximativ egal cu alte intrări create în aproximativ acelaşi timp. Tipul de listă FIFO cu această proprietate este numit *semilistă FIFO* (WFIFO). Intrările nu pot rămîne la nesfîrşit în listă dacă:

1. Tranzacţia care le introduce se termină într-un interval de timp finit.
2. Tranzacţia care le elimină se termină într-un interval de timp finit.
3. Numai un număr finit de tranzacţii elimină o intrare şi apoi se termină cu abort.

Operaţiile permise într-o semilistă sînt aceleaşi cu cele definite pentru liste FIFO adică:

1. *WQEnq(queue,ptr)*: adaugă o intrare la sfîrşitul listei.
2. *WQDeq(queue)*: citeşte şi elimină o intrare de la începutul liste. Dacă lista este vidă, operaţia este blocată pînă cînd lista devine nevidă.

Regulile de sincronizare sînt însă mai puţin restrictive decît cele pentru liste FIFO. Ele sînt:

1. Operaţiile *WQEnq* pot fi executate chiar dacă o altă tranzacţie neterminată a executat alte operaţii *WQEnq*.
2. Operaţiile *WQDeq* pot fi executate chiar dacă o altă tranzacţie neterminată a executat *WQDeq*.
3. O operaţie *WQDeq* se aplică intrării de la începutul listei WFIFO; tranzacţia care a referit această intrare trebuie să se fi terminat.

In exemplele care urmează, o listă WFIFO va reprezentată printr-o secvenţă de litere, cea mai din stînga reprezentînd începutul listei. Literele mici italice simbolizează intrări pentru care operaţiile corespondente *WQEnq* aparţin unor tranzacţii încă neterminate. Literele mari simbolizează intrări care nu au fost eliminate şi pentru care tranzacţiile care le-au introdus au fost terminate. Literele mari italice simbolizează intrări care au fost eliminate prin operaţii WQDeq aparţinînd unor tranzacţii încă neterminate. Cifrele asociate intrărilor simbolizate prin litere italice indică numărul tranzacţiei care a executat acea operaţie.

Presupunem că iniţial lista este vidă. Dacă tranzacţia T_1 execută *WQEnq(WQ,c)* atunci lista conţine:

$\{a^1\}$

Conform regulii 1, o altă tranzacţie T_2 este liberă să creeze o altă intrare:

$\{a^1,b^2\}$

În continuare, T_1 poate să adauge o nouă intrare:

$\{a^1,b^2,c^1\}$

Dacă ambele T_1 şi T_2 se termină normal, starea finală devine:

$\{A,B,C\}$

Presupunem acum că T_3 execută *WQDeq*. Conform regulii 2, o altă tranzacţie T_4 este liberă pentru a elimina două elemente:

$\{A^3,B^4,C^4\}$

Dacă în acest moment T_4 se termină normal iar T_3 execută *abort*, starea finală a listei devine:

$\{A\}$

Se observă că, în acest caz, elementele A şi C s-au inversat, chiar dacă au fost iniţial inserate de aceeaşi tranzacţie. Pe de altă parte, dacă T_3 nu ar fi executat *abort* ci *commit*, deci în condiţii normale, comportarea este asemănătoare unei liste FIFO.

Un alt exemplu ilustrează situaţia cînd o tranzacţie neterminată normal ajunge la începutul listei WFIFO. Presupunem că starea iniţială este:

$\{a^5,b^6\}$

Dacă T_6 executa *commit*, dar T_5 rămîne încă neterminată starea devine:

BUPT

$\{a^5, B\}$

Dacă T_7 elimină un element în acest moment, va fi returnat B, starea devenind:

$\{a^5\}$

după ce T_7 execută *commit*. Pe de altă parte, dacă T_5 execută *commit* după T_6, dar înainte de eliminarea din T_7, va fi returnat A. Din nou se constată că lista WFIFO încearcă să emuleze o listă FIFO obişnuită; în acest caz se permite ca intrări din lista WFIFO, aparţinînd unor tranzacţii neterminate să fie rezervate pînă la terminarea tranzacţiilor cu commit.

Proprietăţile de consistenţă ale unei liste WFIFO pot fi descrise succint prin relaţii de dependenţă. Dependenţele sînt aceleaşi ca şi pentru tipul listă FIFO. Totuşi, spre deosebire de lista FIFO strictă care pretinde ca planificatoarele abstracte să fie ordonabile cu referire la relaţia $\{<_{D1 \cup D3 \cup D5}\}$, lista WFIFO permite formarea de cicluri cu toate relaţiile de dependenţă exceptînd relaţia $<_{D3}$. Permiţînd formarea de cicluri în relaţia $<_{D1}$, se permite astfel intercalarea creerii de intrări de către multiple tranzacţii. Similar, permiţînd formarea de cicluri în relaţia $<_{D5}$, se permite intercalarea ştergerilor de intrări de către mai multe tranzactii.

Pentru a concluziona comparaţia dintre listele FIFO şi WFIFO, trebuie evidenţiat faptul că în listele WFIFO au fost sacrificate două proprietăţi ale listelor FIFO. În primul rînd, nu se respectă ordinea strictă a intrărilor listei. În al doilea rînd, tranzacţiile care operează asupra listelor WFIFO nu mai sînt serializabile. În schimb, sînt mentinute alte proprietăţi esenţiale. De exemplu, o listă WFIFO nu va amîna nedefinit "consumul" unei intrări faţă de "producerea" ei. În plus, lista WFIFO forţează ordonarea acelor tranzacţii care operează printr-un element comun al listei. Acest lucru se asigură prin ordonabilitatea faţă de $\{<_{D3}\}$. Toate aceste modificări măresc într-un grad mare concurenţa, fără a compromite utilitatea specifică a tipului de date.

Pînă acum s-au considerat doar două operaţii posibile în lucrul cu liste de tip coadă (FIFO sau WFIFO). Există însă şi alte operaţii, dintre care unele pot restrînge drastic gradul de concurentă. Un astfel de exemplu este operaţia *QEmpty(q)*. Această operaţie restrînge mult gradul de concurenţă deoarece informaţia care trebuie returnată nu poate fi izolată într-o intrare particulară a listei. De aceea, ca şi operaţia *DumpDir*, operaţia *QEmpty(q)* blochează toate operaţiile ulterioare asupra cozii care modifică informaţia observată. Analog, această operaţie va fi întotdeauna blocată ori de cîte ori exsită un dubiu în rezultatul său (în cazuri de inserări şi eliminări de elemente în tranzacţii neterminate). Nu întotdeauna serializarea tranzacţiilor care apelează operaţia de mai sus este necesară. Să presupunem, de exemplu, că o operaţie *QEmpty* returnează fals, deci există cel puţin o intrare în listă. Dacă urmatoarea operaţie din tranzacţie este *QDeq*, poate surveni o eroare "listă vidă", dacă tranzacţia nu este serializată. Dacă, dimpotrivă, *QEmpty* returnează true, nu este nevoie de serializarea tranzacţiei. Rezultă că o tranzacţie trebuie sau nu serializată în funcţie de rezultatul apelului operaţiei *QEmpty*.

D.Tranzacţii care conţin operaţii asupra mai multor tipuri

Consideraţiile prezentate în subcapitolul anterior relativ la specificaţiile asupra unor tipuri abstracte pot fi extinse pentru a descrie modul în care proprietăţile de consistenţă a mai multor tipuri interacţionează în cazul în care într-o tranzacţie sînt implicate mai multe tipuri.

Trebuie generalizată definiţia de consistenţă aplicată asupra planificărilor abstracte conţinînd operaţii asupra unor tipuri multiple. Presupunem că specificaţiile proprietăţilor de consistenţă pentru tipul Y_1 garantează ordonabilitatea faţă de $\{<_{D1}\}$, pentru tipul Y_2 garantează ordonabilitatea faţă de $\{<_{D2}\}$, etc.

Definiţie.

Setul de planificatoare abstracte consistente faţă de tipurile $Y_1, Y_2, \ldots Y_n$ se defineşte ca setul de planificatoare ordonabile faţă de $\{<_{D1 \cup D2 \cup \ldots \cup Dn}\}$, deci faţă de reuniunea dependenţelor prescrise pentru fiecare tip. Un set de tipuri care satisface aceste proprietăţi este denumit un ***set de tipuri cooperative***.

Exemplul va demonstra insuficienţa ordonabilităţii faţă de dependenţele prescrise pentru fiecare tip în parte. Considerăm următoarea planificare conţinînd tranzacţii care execută operaţii asupra tipurilor discutate anterior (directoare şi liste FIFO).

<T1, QEnq(Q,T), Y>
<T2, QEnq(Q,T), Z>
<T2, InsertDir(D,S), "PDSR")>
<T1, DeleteDir(D,S), "CDR")>

Fie $<_{Dir}$ notaţia pentru relaţia $<_{D2 \cup D4 \cup D8 \cup D9 \cup D11}$ (definită anterior pentru tipul director) şi $<_Q$ notaţia pentru $<_{D1 \cup D3 \cup D5}$ (definită anterior pentru tipul listă FIFO). Deşi planificarea este ordonabilă faţă de relaţia $\{<_{Dir}, <_Q\}$ nu este ordonabilă faţă de relaţia $\{<_{Dir \cup Q}\}$. Pentru a realiza ordonabilitatea, tipurile director şi listă FIFO trebuie să coopereze în prevenirea de cicluri în relaţia compusă.

BUPT

7.4.9.3. Metode de sincronizare

Pentru demonstrarea corectitudinii implementării unui tip trebuie efectuate următoarele etape: (1). caracterizarea setului de planificări de apel legale, adică acele planificări permise de mecanismul de sincronizare utilizat în implementare; (2). realizarea unei corespondenţe între planificarea de apel şi cea abstractă; (3). demonstrarea corectitudinii corespondenţei planificare de apel-planificare abstractă.

Cînd nu se utilizează reordonarea operaţiilor într-o implementare a unei planificări de apel (cazul ideal) atunci planificarea de apel este echivalentă cu planificarea abstractă; în acest caz etapa a doua şi a treia nu mai sînt necesare.

A. Lock-uri specifice tipurilor abstracte

Mecanismul de sincronizare propus în acest subcapitol se bazează pe mecanismul tradiţional de blocare pe bază de lock prezentat in subcapitolul 7.4.4. Există mai multe variante ale acestui mecanism, toate bazîndu-se pe acelaşi principiu: înainte de a opera asupra unui obiect, orice tranzacţie trebuie să obţină un lock pentru obiect; alte tranzacţii nu vor putea accesa obiectul decît după eliberarea lock-ului obţinut de tranzacţia respectivă.

Mecanismul de blocare restricţionează formarea dependenţelor între tranzacţii, restrîngînd setul de planificatoare de apel legale. Ori de cîte ori o tranzacţie este forţată de către o alta să aştepte la un lock, formarea unei dependenţe între cele două tranzacţii este întîrziată pînă cînd prima tranzacţie eliberează lock-ul. Protocolul de blocare în două faze, cel mai cunoscut, impune ca nici o tranzacţie care eliberează un lock să nu mai pretindă un alt lock. Acest lucru are ca efect conversia eventualelor cicluri de relaţii de dependenţă în interblocări. Acestea pot fi detectate şi pentru că nu sînt permise astfel de dependenţe, fiecare tranzacţie poate fi terminată cu abort fără a afecta altă tranzacţie. S-ar putea afirma că protocolul blocării în două faze aplică o politică conservatoare; într-adevăr el întîrzie formarea de orice dependenţe dintr-o relaţie prescrisă, nu numai cele care conduc la cicluri. Totuşi, acesta nu este un dezavantaj propriu-zis, deoarece formarea unor astfel de dependenţe cu transfer de informaţie trebuie întîrziată datorită pericolului unui abort în cascadă.

Dezavantajul cel mai mare al mecanismelor lock tradiţionale este acela că utilizează minimum de informaţii semantice asupra obiectelor manipulate. Cele mai simple scheme lock utilizează un singur tip de *lock*; schemele *lock_S/lock_X* utilizează minimum de informaţie, dar sînt totuşi insuficiente pentru a ţine cont de avantajele dependenţelor specifice tipurilor.

Două observaţii pot fi făcute în ceea ce priveşte dependenţele specifice tipurilor. În primul rînd, ele specifică modul în care operaţiile specifice tipurilor, incluse în tranzacţii, pot fi intercalate. Analog, schema de blocare necesită specificarea claselor *lock*-urilor care corespund operaţiilor asupra acestui tip. În al doilea rînd, dependenţele reflectă datele furnizate ca argumente de către operaţii. De aceea, un obiect *lock* în schema de blocare generalizată constă în două părţi: tipul specific clasei lock-ului şi date specifice instanţierii lock-ului. Notaţia {*LockClass (date)*} este utilizată pentru a reprezenta o instanţiere a unui lock.

După definirea claselor *lock*-urilor pentru un tip definit trebuie furnizată o funcţie booleană care să specifice dacă un nou *lock* (de un anumit tip) este disponibil în funcţie de *lock*-urile deja deţinute de obiect. Pentru similitudine cu notaţiile utilizate în 7.4.4., această funcţie va fi reprezentată printr-o ***tabelă de compatibilitate*** a *lock*-ului. Numai *lock*-uri cerute de alte tranzactii trebuie verificate pentru compatibilitate. O cerere a unui *lock* este compatibilă întotdeauna cu *lock*-urile deţinute de aceeaşi tranzacţie.

B. Blocarea cu lock-uri in cazul directoarelor

Pentru ilustrarea mecanismului de blocare în cazul tipului director se consideră o implementare ideală (care nu reordonează operaţiile) a tipului specificat în paragraful 7.4.9.2. Se presupune, de asemenea, că se asigură sincronizarea la nivelul operaţiilor asupra tipului director (printr-un mecanism oarecare de excludere reciprocă) deoarece mecanismul de blocare se utilizează numai pentru intercalarea corectă a operaţiilor asupra directoarelor din diverse tranzacţii. Mecanismul de blocare şi de excludere reciprocă nu pot fi totuşi complet independente, deoarece excluderea reciprocă trebuie eliminată cînd se aşteaptă un lock.

Deoarece corespondenţa între planificările de apel şi planificările abstracte este imediată pentru implementarea ideală, etapa a doua nu este necesară. Interesează numai prima şi a treia etapă: caracterizarea setului de planificări legale şi comparaţia acestora cu setul de planificări consistente.

Se lucrează cu aceleaşi tipuri de operaţii asupra directoarelor, care pot fi împărţite în trei grupuri:

1. operaţii de *modificare*, care alterează o intrare din director, identificată prin cheia (string) α.

2. operaţii de *căutare*, care detectează prezenta, absenţa sau conţinutul unei intrări particulare din director, identificată prin cheia α.

3. operaţii de *parcurgere*, care observă proprietăţile unui director, care nu pot fi izolate într-o intrare particulară.

Corespunzător acestor grupuri, pot fi definite trei clase de lock:

BUPT

1. {*ModifyDir(α)*};pentru a indica faptul că o tranzacţie incompletă a inserat sau şters o intrare cu cheia α.
2. {*LookupDir(α)*}; pentru a indica faptul că o tranzacţie incompletă observă o intrare cu cheia α.
3. {*DumpDir*}; pentru a indica faptul că o tranzacţie incompletă a executat o operaţie *DumpDir* de listare a întregului director.

Tabela de compatibilitate pentru lock-uri poate fi reprezentată astfel:

Cerere lock \ *Lock Reţinut*	Nimic	ModifyDir(α)	LookupDir(α)	DumpDir
ModifyDir(α)	DA	Nu	Nu	Nu
ModifyDir(α')	DA	DA	DA	Nu
LookupDir(α)	DA	Nu	DA	DA
LookupDir(α')	DA	DA	DA	DA
DumpDir	DA	Nu	DA	DA

Fiecare intrare din această tabelă reflectă specificul relaţiilor de dependenţă pentru directoare. Intrările compatibile reprezintă relaţiile de dependenţă pentru care este permisă formarea de cicluri: de exemplu, pentru linia 2, coloana 2 valoarea este OK deoarece sînt permise cicluri în relaţia de dependenţă $<_{M(\alpha)\rightarrow M(\alpha')}$.

Protocolul utilizat de operaţiile asupra directoarelor pentru alocarea şi eliberarea *lock*-urilor este:

1. Operatiile *InsertDir* şi *DeleteDir* care specifică cheia α obţin un lock {*ModifyDir(α)*}. Dacă operaţia reuşeşte, *lock*-ul este reţinut alocat pînă la sfîrşitul tranzacţiei. Dacă operaţia nu reuşeşte, *lock*-ul este convertit într-un lock {Lookup(α)}, care este reţinut pînă la sfîrşitul tranzacţiei. Această conversie reprezintă o violare a protocolului strict 2PL şi care demonstrează cum se poate îmbunătăţi gradul de concurenţă pentru un protocol specific unui tip.
2. Operaţiile *Lookup* care specifică o cheie α obţin un *lock* {*Lookup(α)*} pentru director pînă la sfîrşitul tranzacţiei.
3. Operaţiile *DumpDir* obţin un *lock* {*DumpDir*} asupra directorului pîna la sfîrşitul tranzacţiei.

C. Blocarea cu lock-uri in cazul listelor

Se prezinta in acest subcapitol specificul folosirii *lock*-urilor pentru liste FIFO, liste WFIFO, si liste cu buffer local.

1.Liste FIFO

Ca şi în exemplul anterior, se presupune o implementare ideală a tipului FIFO inclusiv respectîndu-se excluderea reciprocă la nivelul operaţiilor. Deoarece în cazul acestui tip de date există două operaţii, *QDeq* şi *QEnq*, vor fi definite două clase de lock: {*QEnq(α)*} şi {*QDeq(α)*}. Ca şi în cazul directoarelor, *lock*-urile trebuie să identifice intrarea particulară din listă. Deoarece intrările unei liste FIFO nu se identifică printr-o cheie, se va presupune în continuare că la momentul efectuării unei operaţii *QEnq*, fiecare element din listă contribuie la identificarea ei printr-un identificator unic. Aceşti identificatori corespund celor folosiţi în relaţiile de dependenţă. Astfel, un *lock* {*QEnq(α)*} indică faptul că un element α a fost inserat în listă de o tranzacţie incompletă. Analog, un *lock* {*QDeq(α)*} indică faptul că un element α a fost eliminat din listă. Protocolul folosit în execuţia operaţiilor asupra unor liste FIFO este:

1. Operaţiile *QEnq* trebuie să obţină un *lock* {*QEnq(α)*}, în care α este identificatorul intrării ce trebuie adăugată. Acest lock este reţinut alocat pînă la sfîrşitul tranzacţiei.
2. Operaţiile *QDeq* trebuie să obţină un *lock* {*QDeq(α)*}, în care α este intrarea de la începutul listei. Acest lock este reţinut alocat pînă la sfîrşitul tranzacţiei.

Tabelul următor arată compatibilitatea lock-urilor pentru liste FIFO:

Cerere Lock \ *Lock Reţinut*	Nimic	QEnq(α)	QDeq(α)
QEnq(α)	DA	-	-
QEnq(α')	DA	NU	DA
QDeq(α)	DA	NU	-
QDeq(α')	DA	DA	NU

BUPT

Simbolurile α și α' reprezintă identificatorii a două elemente diferite. Incompatibilitatea dintre operațiile {*QDeq(α)*} și {*QEnq(α)*} asigură interzicerea eliminării intrărilor aparținînd unor tranzacții care nu au executat încă *commit*, înlăturîndu-se astfel situații de abort în cascadă.

2.Liste WFIFO

Se folosește aceeași implementare idealizată a unei liste FIFO și aceleași operații *QEnq* și *QDeq*. Diferența față de tipul precedent constă în tabela de compatibilitate a lock-urilor:

	Lock reținut		
Lock cerut	Nimic	WQEnq(α)	WQDeq(α)
WQEnq(α)	DA	-	-
WQEnq(α')	DA	DA	DA
WQDeq(α)	DA	NU	-
WQDeq(α')	DA	DA	DA

Posibilitatea intercalării operațiilor *WQEnq* de către diferite tranzacții se reflectă în tabelă prin definirea *lock*-urilor {*WQEnq(α)*} și {*WQEnq(α')*} drept compatibile, ca și a *lock*-urilor {*WQDeq(α)*} și {*WQDeq(α)*}. Singura restricție rămîne linia 3, coloana 2, în scopul prevenirii eliminării intrărilor din tranzacții încă neterminate (care nu au executat commit). Aceasta previne formarea de cicluri în relația de dependență prescrisă $<_{E(\alpha)\rightarrow D(\alpha)}$ și previne formarea de abort în cascadă.

Protocolul de blocare pentru operațiile asupra unei liste WFIFO este același ca și în cazul unei liste FIFO. Singura diferență este aceea că dacă operația *WQDeq* nu poate obține un *lock* {*WQDeq(α)*} pentru elementul de la începutul cozii, nu blochează tranzacția ci caută în continuarea listei pînă la obținerea unui lock {*WQDeq(α')*} asupra altui element. Aceasta reflectă proprietatea listelor WFIFO de a permite eliminarea de elemente "din adîncul" listei, în cazul în care elementul de început este implicat într-o tranzacție ce nu a executat commit.

3.Liste FIFO cu buffer local

Considerăm acum o implementare neidealizată a unei liste FIFO, aceea prezentată anterior, în care elementele de inserat în listă se depozitează într-un buffer local care se inserează în listă la terminarea tranzacției prin commit. Pentru a distinge între cele două implementări se va nota implementarea folosind un buffer ca *listă CFIFO*. Pentru listele CFIFO se definesc aceleași operații *CQEnq* și *CQDeq*, dar, în plus, și o operație *CQCommit(cq,tid)* apelată de managerul tranzacției pentru a salva bufferul în momentul cînd tranzacția execută *commit*.

Lista CFIFO poate fi implementată utilizînd o listă FIFO globală, nebufferată, *GlobalQueue*, ca o componentă a implementării și liste locale tranzacțiilor asupra cărora se acționează prin operații *CQxxx*, în regim de excludere reciprocă, dar fără mecanism de blocare, listele nefiind partajate. Operația *CQCommit(cq,tid)* apelează *QEnq(GlobalQueue,ptr)* pentru a plasa fiecare din intrările listei locale în lista FIFO globală.

Primul apel *QEnq* trebuie să poată obține un lock {*QEnq*(α_i)} asupra listei FIFO globale; alocarea acestui lock permite unei tranzacții să obțină acces exclusiv la lista globală, deoarece *lock*-ul {*QEnq*(α_i)} necesar pentru fiecare intrare particulară de inserat este compatibil cu cel precedent.

Implementarea CFIFO permite planificări de apel în care operațiile *CQEnq* din diferite teanzacții pot fi intercalate. Amînarea operațiilor *QEnq* asupra listei FIFO globale pînă la execuția operației *commit* transformă planificările de apel în planificări abstracte fără intercalare de operații.

Dacă lista CFIFO globală nu este vidă, prima dintre intrări trebuie eliminată și returnată. Pentru a preveni cicluri în relația $<_{E(\alpha)\rightarrow D(\alpha)}$, este necesară blocarea, dacă intrarea este reținută de o tranzacție care nu a executat încă *commit*. Dacă lista *GlobalQueue* este vidă, orice intrări din bufferul local trebuie transferate în *GlobalQueue* printr-un proces identic cu cel din *CQCommit*. Prima intrare din bufferul local de intrări al tranzacției poate apoi fi eliminată din *GlobalQueue* și returnată. Deplasarea intrărilor buferate în bloc în *GlobalQueue* interzice formarea de cicluri în relațiile prescrise $<_{E(\alpha)\rightarrow E(\alpha')}$ și $<_{D(\alpha)\rightarrow D(\alpha')}$. Un al treilea caz apare atunci cînd și lista *GlobalQueue* și bufferul sînt vide; în acest caz operația trebuie blocată pînă cînd o altă tranzacție inserează o intrare în *GlobalQueue* și execută *commit*.

Operațiile descrise nu sînt însă suficiente pentru a implementa *CQDeq*. Dificultatea constă în aceea că *QEnq(GlobalQueue)* va fi blocată dacă lista globală este vidă, lucru care trebuie evitat pentru a beneficia de avantajele bufferului local. De aceea, mai trebuie introdusă o operație, *QDeqNE* necesară în cazul prezentat mai sus. Dacă se poate elimina o intrare din lista globală, *QDeqNE* alocă un lock {*QDeq(α)*} asemenea cu *QDeq*. În caz contrar și dacă există intrări în bufferul local, acestea sînt deplasate în lista globală, după care se apelează operația standard *QDeq* pentru a elimina și returna prima intrare din lista globală. În acest caz, tranzacția reține alocat un lock {*QDeq(α)*} asupra listei pînă la terminare.

BUPT

Capitolul 8

CONTROLUL INTERBLOCĂRII OBIECTELOR

Problemele care se pun în legatură cu controlul interblocării în sistemele distribuite sunt asemănătoare cu cele pentru sistemele centralizate. În sistemele distribuite, însă, datorită dispersiei informaţiilor pe mai multe noduri, interblocările sunt şi mai greu de prevenit, evitat şi detectat.

Se pot distinge două tipuri de interblocări în sistemele distribuite: interblocările rezultate ca urmare a utilizării resurselor în acces exclusiv şi interblocările de comunicare, care apar cînd un set de procese aşteaptă circular primirea unui mesaj. În secţiunile următoare se va trata primul caz de interblocări, cosiderînd că, pe de o parte, al doilea caz poate fi inclus în primul dacă se consideră canalele de comunicaţie ca resurse şi, pe de altă parte, se ţine cont de aprecierea ([Tan93], [Tan96], [Gos91]) că o astfel de comunicaţie circulaăa apare destul de rar în aplicaţii (paradigma de interacţiune a proceselor cea mai des întîlnită este cea client-server).

8.1. Metode de control al interblocării

În principiu, toate metodele de control al interblocării cunoscute în sistemele centralizate sunt valabile şi pentru sistemele distribuite. Astfel, metoda prevenirii interblocării prin ordonarea resurselor poate fi utilizată prin definirea unei relaţii de ordine peste evenimentele sistemului. Se asignează tuturor resurselor sistemului distribuit numere unice iar un proces poate cere o resursă i, de pe orice procesor, dacă şi numai dacă nu deţine o resursă cu număr mai mare sau egal cu i. Similar, algoritmul bancherului poate fi utilizat într-un sistem distribuit dacă se desemnează unul dintre procesele din sistem ca proces central care menţine toate informaţiile necesare algoritmului; fiecare cerere de acces la resursă va fi adresată acestui proces. Totuşi, în afară de faptul că, fiind centralizată, metoda poate conduce la sufocarea procesului central, ea prezintă acelaşi dezavantaj ca şi în sistemele centralizate: impune cunoaşterea în avans a resurselor utilizate, ceea ce, în sistemele distribuite este arareori posibil. Ca atare, în sistemele distribuite aceste metode nu se folosesc. Se folosesc, în schimb, metodele de prevenire a interblocării care se bazează pe tranzacţii atomice şi pe ordonarea tranzacţiilor prin asocierea de mărci de timp şi metode de detecţie a interblocării.

Utilizarea tranzacţiilor atomice facilitează procedura de distrugere şi reluare a proceselor implicate intr-o interblocare; în cazul tranzacţiilor atomice, acestea pot fi readuse în starea de înaintea începerii tranzacţiei, pe cînd reluarea unor procese simple de la o stare anterioară necesită mai multe complicatii.

8.2. Prevenirea interblocărilor

În sistemele distribuite în care sunt implementate tranzacţii atomice şi timp global sunt folosiţi în practică doi algoritmi pentru prevenirea interblocărilor. Ambii se bazează pe ideea asignării fiecărei tranzacţii, în momentul în care începe, a unei mărci de timp. Fiecărei tranzacţii i se mai asignează şi o prioritate, care poate fi numărul de proces; aceasta urmează să fie folosită în cazul în care două tranzacţii au aceeaşi marcă de timp.

Cei doi algoritmi sunt:

(1) algoritmul *wait_die* care se bazează pe o tehnică nepreemptivă. Cînd o tranzacţie T_i cere accesul la o resursa deţinută la momentul respectiv de T_j, T_i va aştepta alocarea resursei numai dacă are o marcă de timp mai mică decît T_j (aceasta înseamnă că T_i este o tranzacţie mai veche în sistem decît T_j); în caz contrar T_i este abortată. În figura 7.5.2.1 se prezintă un exemplu de funcţionare al acestui algoritm.

(2) algoritmul *wound_die* care se bazează pe o tehnică preemptiva. cînd o tranzacţie T_i cere accesul la resursa deţinută la momentul respectiv de T_j, T_i va aştepta alocarea resursei numai dacă are o marca de timp mai mare decît T_j (aceasta înseamnă că T_i este o tranzacţie mai noua în sistem decît T_j); în caz contrar T_j este abortată, iar T_i continuată (fig 7.5.2.2).

Ambii algoritmi evită amînarea la infinit a tranzacţiilor, cu condiţia ca, la reluarea tranzacţiilor după abortare, să se pastreze aceeaşi marcă de timp. Deoarece mărcile de timp au valori crescatoare, o tranzacţie abortată va deveni, la un moment, cea mai veche din sistem şi va avea cea mai mică marca de timp din sistem. Există diferenţe semnificative în ceea ce priveşte modul de operare al algoritmilor în acest sens. Astfel: (1) În algoritmul *wait_die*, o tranzacţie mai veche aşteaptă ca o tranzacţie mai nouă să elibereze resurse, pe cînd în algoritmul *wound_die* o tranzacţie mai veche nu aşteaptă niciodată la o tranzacţie mai nouă. (2) În algoritmul *wait_die*, o tranzacţie mai nouă este abortată cînd cere accesul la o resursa deţinută de o tranzacţie mai veche. În mod normal, programul va încerca din nou realizarea tranzacţiei şi, din acelaşi motiv, ea poate fi abortată de mai multe ori; în contrast, în algoritmul *wound_die*, o tranzacţie nouă va aştepta eliberarea resursei de catre tranzacţia mai veche. În schimb, algoritmul *wound_die* abortează o tranzacţie nouă cînd o alta mai veche cere aceeaşi resursă. Ca atare, se poate aprecia că algoritmul *wound_die* necesită, în general, mai puţine operaţii abort.

BUPT

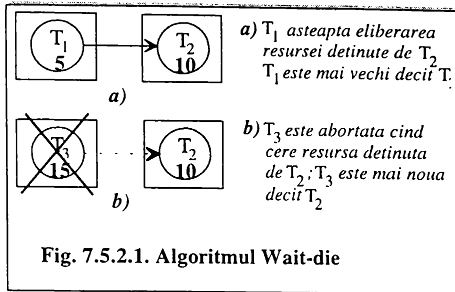


Fig. 7.5.2.1. Algoritmul Wait-die

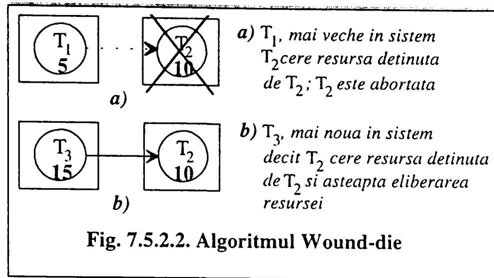


Fig. 7.5.2.2. Algoritmul Wound-die

8.3. Detecția interblocărilor în sisteme distribuite

Prevenirea interblocării poate genera abortarea unor tranzacții care uneori nu generează interblocare. Pentru a înlatura acest dezavantaj, se folosesc *algoritmi de detecție* a interblocării. Detecția interblocării în sisteme distribuite constituie un subiect al multor cercetari, propunîndu-se diverşi algoritmi, pentru a căror certitudine s-au facut demonstrații, dar pentru care însă, ulterior s-au gasit şi contraexemple. Corectitudinea unui algoritm de detecție depinde de două condiții: (1) *condiția de progres*: fiecare interblocare trebuie detectată; (2) *condiția de siguranta*: dacă s-a detectat o interblocare, atunci ea trebuie să existe intr-adevar. În secțiunea 7.5.3.3 se arată că, datorită întîrzierilor în transmisia mesajelor, într-un sistem distribuit pot fi detectate interblocări false.

8.3.1 Modelul de lucru

Teoria detecției interblocărilor poate fi aplicată direct sistemelor de obiecte distribuite. Ca atare, în această secțiune va fi descris modelul de lucru folosit în secțiunile următoare şi aplicat sistemelor de obiecte distribuite. Acest model constă într-un set de N noduri, S_1, S_2,... S_N conectate intr-o rețea; se consideră transmisia mesajelor ca sigură iar nodurile complet conectate. În fiecare nod există un sistem de obiecte, centralizat, component al sistemului de obiecte distribuite. Există M tranzacții T_1, T_2,... T_M în execuție în contul sistemelor de obiecte distribuite. O tranzacție adresează cererile de execuție a unei operații asupra obiectului local (fişiere, înregistrari, bază de date locală) unui *manager central al tranzacțiilor* sau *controler*. Există cîte un controler C_i pentru fiecare nod S_i. O cerere de acces la obiect poate consta, de exemplu, într-o operație de scriere, citire sau blocare a obiectului. O tranzacție este blocată din momentul în care adresează o cerere managerului central şi pînă în momentul în care controlerul oferă accesul la obiect (cînd tranzacția devine activă). Cererea se poate referi la un obiect local sau la un obiect situat la distanță, caz în care tranzacția devine distribuită.

O tranzacție distribuită T_i se implementează prin mai mulți agenți ai tranzacției, t_{ij}, cîte unul pentru fiecare nod S_j. Cînd o tranzacție cere acces la un obiect situat la distanță, se creează un agent care are rolul de a realiza cererea la obiect, accesul la obiect şi eliberarea obiectului; agentul este distrus cînd cererea la distanță a fost îndeplinită. Pentru simplitate, agentii tranzacțiilor se vor putea nota şi cu un singur indice, t_i semnificînd deci al i-lea agent.

În cazul în care agentul t_{ij} cere accesul la un obiect nelocal, gestionat de controlerul C_m, controlerul C_j transmite cererea la agentul t_{im}, via controlerul C_m. Cînd t_{im} obține accesul la obiect de la C_m trimite un mesaj la t_{ij}, anunțîndu-l că accesul a fost obținut. Cînd agentul tranzacției T_i nu mai are nevoie de accesul la obiectul controlat de C_m, ii comunică agentului t_{im} care este responsabil pentru eliberarea resursei la C_m. Astfel toate cererile între noduri sunt transformate în cereri între agenți ai aceleiaşi tranzacții.

Modelul matematic al cererilor de acces la obiecte ale tranzacțiilor este un graf, *graful dependentelor tranzacției (TWFG, transaction wait_for graph)*. Nodurile acestui graf se asociază proceselor agent ale tranzacțiilor. Arcele reprezintă relații de blocare între procesele agent. Un nod care are un arc ce iese din acel nod corespunde unui proces agent blocat. Mai precis, într-un graf TWFG există un arc de la agentul t_{ij} la agentul t_{kj} dacă controlerul C_j are o cerere de la t_{ij} referitoare la o resursa deținută de t_{kj}; un astfel de arc se numeşte *arc intracontroler*. Un arc de la agentul t_{ij} la agentul t_{im} există dacă t_{ij} aşteapăt un mesaj de permisiune acces la obiect de la t_{im}; un astfel de arc se numeşte *arc intercontroler*.

O structură ciclică în acest graf indică o interblocare. Definiția precisă a termenului de *structură ciclică* depinde de modelul de interblocare considerat (vezi secțiunea 7.5.3.2.2). Un prim exemplu este prezentat în fig.7.5.3.1. Există 5 tranzacții, implementate prin 7 procese agent. Prezența ciclului t_{21} -> t_{31} -> t_{32} -> t_{42} -> t_{22} -> t_{21} indică o interblocare dacă tranzacției T_2 ii sunt necesare ambele obiecte (1 şi 2) pentru a continua prelucrările; dacă, însă îi este necesar numai un singur obiect dintre cele două, nu apare interblocare.

Principala problemă în sistemele distribuite este construcția grafului global TWFG. O serie de algoritmi propun menținerea în fiecare nod a grafurilor TWFG locale. Pentru a se demonstra că nu apare interblocare, trebuie arătat că graful global obținut prin reuniunea tuturor grafurilor TWFG locale nu are structuri ciclice.

BUPT

8..3.2 Clase de algoritmi pentru detecția interblocării

Există numeroase metode de a organiza graful TWFG într-un sistem distribuit. Clasificarea algoritmilor de detecție a interblocării se face în funcție de modul în care se construieste graful TWFG.

8..3.2.1.Clasificarea algoritmilor

Detecția interblocării poate fi realizată utilizînd algoritmi cu caracter *centralizat, ierarhic și distribuit. Algoritmii centralizați* construiesc graful global TWFG prin reuniunea grafurilor locale. Acesta este construit și menținut de către un singur proces: *procesul coordonator* al detecției interblocărilor. Algoritmii centralizați prezintă dezavantajele oricărei metode centralizate: vulnerabilitatea procesului central la avarii și traficul ridicat de mesaje. Unui detector centralizat îi este necesar un interval de timp considerabil pentru colectarea și asamblarea grafurilor locale TWFG; ca atare un alt dezavantaj este și faptul că o interblocare rămîne nedetectată un interval de timp relativ mare.

O alternativă la menținerea tuturor informațiilor necesare detecției interblocării într-un singur proces este de a distribui aceste informații între mai multe procese și de a impune o organizare ierarhică a facilităților de detecție, obținandu-se o altă categorie de algoritmi, *algoritmii ierarhici.*

O a treia categorie de algoritmi sunt *algoritmii complet distribuiți*; în cazul acestora problemele principale sunt menținerea consistenței informațiilor, pentru a nu se detecta interblocări false și pentru nu se detecta aceeași interblocare în mai multe sisteme. Costul unui algoritm distribuit se masoară în numărul de mesaje schimbate între nodurile implicate în interblocare și timpul de calcul.

8..3.2.2 Modele de interblocare

În funcție de aplicație, se pot permite diverse tipuri de cereri de operații asupra obiectelor. De exemplu, o tranzacție poate cere o combinație de operații asupra mai multor obiecte (obiectul a *și* obiectul b *sau* obiectul c) sau poate cere accesul numai la un singur obiect. Aceasta distincție permite o clasificare a algoritmilor distribuiți de detecție, conform complexității cererilor de acces pe care aceștia le permit.

Cel mai simplu model este cel în care o tranzacție poate avea în așteptare la un moment doar o singură cerere de accces la un obiect. Aceasta înseamnă că ordinul de ieșire al graficului TWFG (numărul maxim de arce ce pot iesi dintr-un nod) este 1. A detecta interblocarea pentru un astfel de model înseamnă a gasi un ciclu în graful global TWFG.

Modelul AND permite unei tranzacții să ceară la un moment dat accesul la un set de obiecte. O tranzacție va fi blocată dacă nu primește accesul la toate obiectele; de aceea modelul se numește modelul AND. Si în acest model, detecția interblocării înseamnă a găsi un ciclu în graful global TWFG.

O definiție mai riguroasă a interblocării în modelul AND a fost dată de Chandy și Misra ([CM82]). Un proces t_i este dependent de un proces agent t_j dacă există o secvență seq=t_i, t_{i1},... t_{im}, t_j de procese agent blocate și fiecare proces din seq, cu excepția primului, deține o resursă la care așteaptă procesul anterior din seq; se spune că *t_i este dependent local de t_j* dacă toate procesele agent din seq aparțin aceluiași controler. În acest caz, t_i este interblocat dacă este dependent de el însuși sau de un agent care este dependent de el însuși; în ambele cazuri interblocarea apare numai dacă există un ciclu de procese agent blocate.

Algoritmii de detecție a interblocării pentru modelul AND stabilesc că există interblocare dacă și numai dacă există cicluri de procese blocate. Aceasta condiție nu impune însă, că dacă un agent t_i este interblocat el va fi și detectat ca interblocat. De fapt, dacă t_i este interblocat, dar nu face parte dintr-un ciclu de procese blocate, t_i nu va fi niciodată declarat interblocat. Un exemplu este agentul t_{51} din figura 7.5.3.1.

Interblocarea în modelul cu un singur obiect poate fi definită asemănător, punîndu-se în plus restricția ca un proces agent să poată avea în așteptare numai o singură cerere (un singur arc care iese din nodul procesului agent). De aici rezultă că modelul AND este o generalizare a modelului cu un singur obiect. Pentru modelul AND au fost propuși și discutati prin exemple și contraexemple mulți algoritmi de detecție ([Gos91], [CM82][CMH83], [GS80], [MM79]).

Modelul OR permite unei tranzacții să ceară un singur obiect dintr-un set de obiecte; un exemplu adecvat este o cerere de citire dintr-un sistem de obiecte replicate: citirea poate avea loc la orice copie disponibilă. În modelul OR, apariția unui ciclu este insuficientă pentru detecția interblocării. În același exemplu din fig.1, dacă se consideră cereri de tip OR, tranzacția T_2 nu este blocată, deoarece t_{11} nu are arce de ieșire și dupa ce t_{11} eliberează obiectul, T_2 poate continua. Definiția interblocării pentru modelul OR a fost dată în [CM82]. În acest model se definește un proces agent blocat, un proces care are o cerere OR în așteptare; fiecărui proces blocat i se asociază un set de procese, denumit *set dependent*; un proces blocat își reia execuția dacă primește permisiunea de acces la un obiect de la un proces din setul său dependent. Un set de procese S este *interblocat* dacă: (1) toate procesele din S sunt blocate; (2) setul dependent al fiecărui proces din S este un subset al lui S; (3) nu există în tranzit mesaje de permisie acces la obiect între procesele din S. Un proces este *interblocat* dacă aparține unui set interblocat.

În termeni TWFG, o interblocare în modelul OR este indicată de prezența unui nod *knot*; prin definiție un nod este de tip *knot* dacă oricare ar fi w cu proprietatea că w poate fi atins din v atunci și v poate fi atins din w. Astfel, detecția interblocării în modelul OR se reduce la a gasi nodurile knot din graful WTFG. În secțiunea 8..3.5.2 se prezintă un algoritm de detecție pentru acest model.

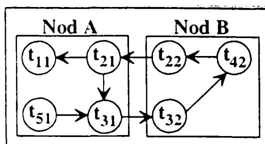


Fig. 7.5.3.1. Graf TWFG.

Modelul AND_OR este o generalizare a celor două modele anterioare. În modelul AND_OR se poate specifica orice combinație *and* și *or* de cereri de acces la obiect; de exemplu, (a *and*(b *or* c)) *or* d. Nu există o structură din teoria grafurilor care să descrie situația de interblocare în modelul TWFG. În principiu, detectarea

BUPT

interblocării în modelul AND_OR se face prin aplicarea repetată a testului pentru modelul OR, exploatînd astfel faptul că interblocarea este o proprietate stabilă (adică nu dispare de la sine).

Modelul (n^K) permite unei tranzacţii să ceara K resurse dintr-un set de n resurse disponibile. Este o generalizare a modelului AND_OR; este adevarată şi reciproca, adică modelul AND poate fi reprezentat în modelul (n^K) ca (n^n), iar modelul OR ca (1^n). O definiţie şi un algoritm de detecţie a interblocării pentru acest model au fost date în [BT84].

Modelul fără restricţii nu impune nici o structurare a cererilor de acces la obiecte; singura condiţie impusă este stabilitatea interblocării. O teorie generală a acestui model poate fi gasită în [CL85], [CM86].

8..3.2.3 Clase de algoritmi distribuiti pentru detecţia interblocării

Există patru mari clase de algoritmi distribuiti pentru detecţia interblocării: *algoritmi care transmit caile, algoritmi care urmaresc arcele, algoritmi cu difuzie şi algoritmi care stabilesc starea globala.*

Algoritmii care transmit caile (path-pushing) constituie prima clasa de algoritmi proiectaţi pentru detectarea interblocării. Acesti algoritmi construiesc explicit graful global TWFG. Ideea de bază este de a construi o forma simplificată a grafului TWFG în fiecare nod. În acest scop, fiecare nod transmite graful sau local la un anumit număr de vecini de fiecare dată cînd se execută testul de interblocare. Dupa actualizarea structurilor de date locale, de fiecare nod, grafurile locale TWFG sunt transmise mai departe, procedura repetîndu-se pînă unul dintre noduri are o imagine suficient de completă pentru a stabili existenţa sau absenţa interblocării. Deoarece caracteristica principală a acestor algoritmi este transmiterea căilor din graful TWFG ei sunt înglobaţi sub denumirea de algoritmi *path-pushing.*

Algoritmii care urmăresc arcele (edge-chasing) detectează prezenţa unui ciclu într-un graf TWFG prin propagarea unor mesaje de probă în lungul arcelor grafului. Mesajele de probă sunt distincte de mesajele cerere a unui obiect sau mesajele de permisiune acces la obiect. Cînd un iniţiator al unei astfel de proceduri recepţionează un mesaj de probă, înseamnă că există un ciclu în acel graf. Un exemplu de algoritm din aceasta clasă este prezentat în [CM82] şi descris în secţiunea 8..3.5.1.

Algoritmii cu difuzie au ca idee de bază activarea unei prelucrări cu difuzie, de exemplu, de un manager de tranzacţii care suspectează o interblocare. Caracteristic prelucrării este faptul că graful global TWFG este reflectat implicit în structura prelucrării; graful global nu este construit explicit. Prelucrarea cu difuzie se împrăstie prin transmiterea de mesaje *query* şi se restrînge prin recepţia de mesaje *reply*; în momentul cînd prelucrarea cu difuzie a ajuns înapoi la iniţiator, algoritmul se termină şi iniţiatorul stabileşte o interblocare. Un exemplu de algoritm din această clasă este prezentat în [CMH83] şi descris în secţiunea 8..3.5.2.

Algoritmii care stabilesc starea globala se bazează pe rezultatele obţinute de Chandy şi Lamport în [CL85]. Noţiunea cheie este starea globală consistentă a sistemului care poate fi determinată fără suspendarea temporară ("îngheţarea") a prelucrărilor aplicaţiei. Chandy şi Lamport arată cum se poate obţine o stare globală consistentă a unui sistem distribuit prin propagarea unor marcatori de-a lungul canalelor sistemului; o astfel de stare este denumită o vedere (snapshot) a sistemului şi poate fi examinată off-line în ceea ce priveşte interblocările.

8..3.3 Algoritmi centralizaţi pentru detecţia interblocării

În algoritmii centralizaţi pentru detecţia interblocării, graful global TWFG se construieşte prin reuniunea tuturor grafurilor TWFG locale. El este menţinut de catre un proces central, denumit *coordonatorul detecţiei interblocării.* Datorită întîrzierilor mesajelor în sistemele distribuite, se disting două tipuri de grafuri TWFG: *graful real* care descrie starea reală a sistemului distribuit, din orice moment, stare care însă nu poate fi cunoscută şi *graful construit* care este o aproximare a grafului real, realizat de coordonator în timpul execuţiei algoritmului. Evident, graful construit trebuie generat astfel încît atunci cînd algoritmul de detecţie este executat, rezultatele raportate să fie corecte în sensul condiţiilor prezentate în secţiunea 8..3.3. Constructia grafului poate avea loc în următoarele trei moduri: (1) ori de cîte ori un nou arc este adaugat sau eliminat dintr-un graf local; (2) periodic, dupa un număr de modificări în graf; (3) ori de cîte ori coordonatorul execută algoritmul de detectare a unui ciclu în graf.

Consideram modelul cererii de acces la un singur obiect. cînd se execută algoritmul de detectare a unui ciclu în graf, coordonatorul analizează graful global; dacă se gaseste un ciclu, se selectează o tranzacţie victimă care se abortează. Există două situaţii care se pot solda în mod eronat cu abortarea tranzacţiilor:

(1) detectarea ciclurilor false care pot apare în graful TWFG global. Exemplul din figura 7.5.3.3.1 ilustrează acest lucru. Ciclul fals apare datorită receptiei mesajului de inserare a unui arc înainte de recepţia celui de ştergere. Detectarea ciclurilor false de acest tip poate fi ocolită dacă se foloseşte o marcă de timp pentru fiecare mesaj şi ordonarea evenimentelor folosind algoritmul lui Lamport. Astfel, cînd coordonatorul recepţionează un mesaj cu marca de timp TS care conduce la o situaţie de interblocare, poate trimite un mesaj broadcast care conţine marca de timp catre toate nodurile, pentru că acestea, dacă au cumva mesaje cu mărci de timp mai mici, să le trimită imediat. Metoda necesită însă o circulaţie excesiva de mesaje.

(2) abortări necesare ale tranzacţiilor pot apare şi dacă există într-adevar o interblocare şi a fost aleasă o tranzacţie victima, dar în acelaşi timp una dintre tranzacţii este abortată din alte motive (de exemplu şi-a depaşit timpul maxim de existenţă).

Algoritmul centralizat următor [SP88] detectează toate interblocările fără a detecta şi interblocări false. Cînd o tranzacţie T_i din nodul A are un obiect deţinut de T_j în nodul B, se trimite către coordonator un mesaj cu marca de timp TS_n şi se inserează arcul (T_i, T_j,TS_n) în graful local al lui A. Arcul (T_i, T_j,TS_n) se inserează şi în graful local al lui B numai dacă T_j a recepţionat mesajul ce conţine cererea şi nu poate elibera resursa. O cerere de la T_i la T_j în acelaşi nod este tratată în aceeaşi manieră dar nu se mai asociază o marcă de timp arcului (T_i, T_j). Algoritmul de detecţie al interblocării este următorul: (1) Coordonatorul trimite un mesaj de iniţiere la fiecare nod din sistem; (2) la primirea acestui mesaj, toate nodurile trimit grafurile lor locale coordonatorului. Fiecare dintre aceste noduri conţine graful

BUPT

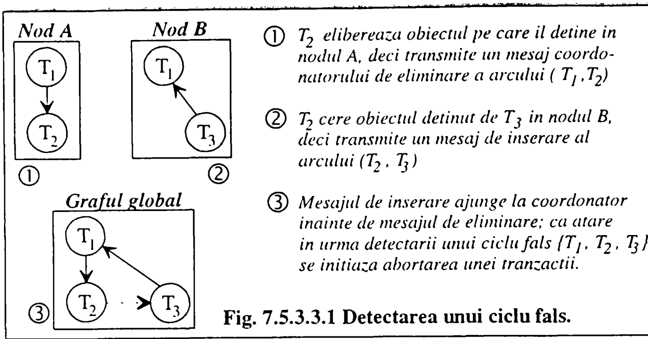


Fig. 7.5.3.3.1 Detectarea unui ciclu fals.

real care reprezintă o stare instantanee a nodului, dar aceasta nu este sincronizată în raport cu celelalte noduri ale sistemului; (3) cînd coordonatorul a primit toate raspunsurile, construieşte graful global astfel: (a) Graful total conţine cîte un nod pentru fiecare tranzacţie din sistem; (b) Graful are un arc (T_i, T_j) dacă şi numai dacă: (i) există un arc (T_i, T_j) intr-unul din grafurile locale; (ii) un arc (T_i, T_j, TS_n) apare în mai mult decît un graf local. Dacă există un ciclu în graful local astfel construit, atunci sistemul este interblocat; dacă nu există un astfel de ciclu, atunci sistemul nu este interblocat în momentul începerii execuţiei algoritmului.

8..3.4. Algoritmul ierarhic pentru detecţia interblocării

Ca şi în cazul algoritmului centralizat, fiecare nod menţine graful sau local. În contrast însă cu algoritmii centralizati, graful global TWGF este distribuit într-un anumit număr de noduri coordonatoare. Acestea sunt organizate într-un arbore, în care fiecare frunză conţine graful local al unui singur nod din sistemul distribuit; celelalte noduri (care nu sunt frunze) ale arborelui conţin informaţii relevante ale grafurilor coordonatorilor din subarborele respectiv. Fie trei noduri coordonatoare A, B şi C, C fiind cel mai tînar stramoş comun al lui B şi C. Presupunem că nodul T_i apare în graful local al lui A şi B; atunci T_i trebuie să apară în următoarele grafuri: (1) coordonatorului C; (2) fiecărui coordonator din calea de la C la A; (3) fiecărui coordonator din calea de la C la B. În plus, dacă T_i şi T_j apar în graful TWFG al coordonatorului D şi există o cale de la T_i la T_j în graful unui fiu al lui D, atunci trebuie să existe un arc (T_i, T_j) în graful lui D. Dacă există un ciclu în oricare dintre aceste grafuri, atunci există interblocare. Pentru a ilustra acest algoritm se consideră exemplul din fig. 7.5.3.4. Deoarece T_2 şi T_3 apar în A şi B, atunci apar şi în C; deoarece există în A o cale de la T_2 la T_3, atunci se include în C un arc (T_2, T_3). Analog, deoarece există o cale de la T_3 la T_2 în B, arcul (T_3, T_2) se include în C. Se observă că graful lui C conţine un ciclu, ceea ce implică prezenţa interblocării.

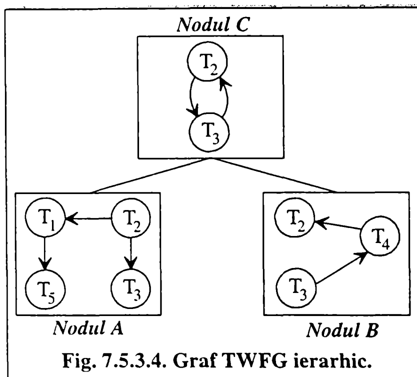


Fig. 7.5.3.4. Graf TWFG ierarhic.

8..3.5. Algoritmi distribuiţi pentru detecţia interblocării

În aceasta secţiune vor fi prezentaţi doi algoritmi distribuiţi: un algoritm din clasa algoritmilor edge-chasing pentru detecţia interblocării în modele AND şi un algoritm din clasa algoritmilor cu difuzie pentru detecţia interblocării în modele OR.

8..3.5.1. Algoritm edge-chasing pentru detecţia interblocării în modele AND

Algoritmul următor a fost propus de Chandy şi Misra în [CM82] şi [CMH83] pentru detecţia interblocării în modele AND. Se presupune că pentru detecţia interblocării fiecare controler execută o copie a algoritmului. Pentru a determina cînd un agent al unei tranzacţii în aşteptarea accesului la un obiect este interblocat, controlerul său iniţiază transmisia unor *mesaje de probă*. Iniţierea mesajelor de probă poate fi facută pentru mai multe tranzacţii.

Un *mesaj de probă* constă într-un triplet (i, j,k) în care i specifică iniţiatorul, iar j şi k că mesajul a fost transmis pe arcul intercontroler (t_j, t_k). Un controler trimite un mesaj de probă (i, j, k) dacă sunt adevarate următoarele condiţii: (1) t_j este în aşteptare la o resursă; (2) t_j asteaptă la t_k; (3) t_i este dependent de t_j. Mesajele de probă recepţionate de un controler pot fi aşteptate sau ignorate. Un mesaj de probă (i, j, k) este aşteptat dacă t_k este în aşteptare la o resursă. Este evident că, dacă un controler al unui agent t_i primeste mesajul de probă (i, j, i), oricare ar fi j, rezultă că t_i este interblocat.(fig. 7.3.5.1.)

Algoritmul este descris în continuare:

iniţierea mesajelor de probă de catre controlerul agentului de tranzacţie t_i:

if t_i este dependent local de el insusi ⟶
  declara t_i interblocat
[] t_i nu este dependent local de el insusi ⟶
  fa arcele de iesire (t_j, t_k) ale lui t_i ⟶ *send* (i, j, k)
  af
fi

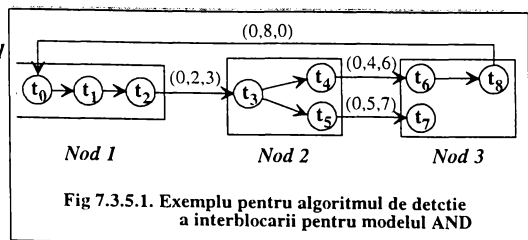


Fig 7.3.5.1. Exemplu pentru algoritmul de detctie a interblocarii pentru modelul AND

BUPT

controler care recepționează un mesaj de probă::

if mesajul de proba (i, j, k) este acceptat ⟶

if k=i ⟶ declara t_i interblocat

[] *fa* arcele de iesire (t_p, t_q) ale lui t_k ⟶ *send* (i, p, q) *af*

fi

fi

Pentru a întrerupe o interblocare există mai multe soluții. O soluție este ca procesul care a inițiat mesajele de probă să se autodistrugă. Pentru algoritmul prezentat mai sus, această soluție nu este convenabilă, deoarece algoritmul poate fi executat simultan de mai multe procese, deci se pot autodistruge mai multe procese. O altă soluție este ca fiecare proces să-și adauge identitatea în cadrul mesajului de probă, astfel încît la returnarea mesajului la inițiator lista să fie completă. Inițiatorul poate alege pentru distrugere procesul cu numărul cel mai mare, trimitîndu-i un mesaj să se autodistrugă. Chiar dacă algoritmul de detecție a interblocării a fost inițiat din mai multe noduri, pentru autodistrugere se va alege același proces.

8..3.5.2 Algoritm cu difuzie pentru detecția interblocării în modele OR

Algoritmul următor a fost propus de Chandy, Misra și Haas în [CMH83] pentru detecția interblocării în modele OR. Un proces agent poate determina cînd este interblocat prin inițierea unei prelucrări cu difuzie. Mai multe procese pot iniția prelucrări cu difuzie în același timp. Mesajele trimise în cadrul unei prelucrări cu difuzie au formele: *query (i, j, k)* și *reply (i, j, k)*, unde i specifică procesul inițiator p_i, j specifică emitatorul p_j, iar k receptorul p_k. Există cel mult un mesaj de forma *query (i, j, k)* și cel mult o replică *reply (i, j, k)* la mesajul *query (i, j, k)*. Un mesaj inițiator trimite mesaje query la toate procesele din setul său dependent (vezi fig. 7.5.3.2.2). Ideea de bază este ca un proces blocat, la primirea unui mesaj query, trebuie să propage mesajul în întregul său set dependent, dacă nu a facut-o deja. Dacă există o secvență de procese blocate p_i, ... , p_j astfel încît fiecare proces din secventă, cu excepția primului, este în setul dependent al procesului anterior din secventă, un mesaj *query* inițiat de p_i se va propaga la p_j. La primirea unui mesaj *query* sau *reply*, un proces p_k, dacă este activ, îl ignoră; dacă este blocat în așteptare la un obiect, există mai multe posibilități. Dacă p_k a primit primul mesaj *query* de la un inițiator i (primul mesaj *query* se numește *de angajare*) îl propagă la toate procesele din setul său dependent și memorează numărul de mesaje într-o variabilă locală, fie ea *nu[i]*; o altă variabila locală *wait[i]* specifică faptul că p_k a fost continuu blocat de la primirea mesajului query. Dacă p_k primeste alte mesaje *query*, raspunde cu *reply* dacă *wait[i]* are valoarea adevarat; în caz contrar, dacă *wait[i]* este fals, ignoră mesajul *query*. cînd p_k primeste un mesaj *reply* și *wait[i]* este adevarat, decrementează *num[i]*. Un proces p_k trebuie să raspunda la un mesaj de angajare numai cînd a recepționat un mesaj *reply* pentru fiecare cerere pe care a propagat-o, deci dacă *num[i]*=0. Cînd un proces p_k inițiază o prelucrare cu difuzie, el trimite un mesaj *query (k, k, j)* la fiecare proces din setul său dependent și setează *num[k]* la numărul de mesaje trimise. Dacă inițiatorul p_k primește răspunsuri *reply* la toate mesajele *query* trimise, atunci inițiatorul este interblocat. Algoritmul este în continuare:

initierea unei prelucrări cu difuzie de catre un inițiator p_k ::

fa p_j din setul dependent S al lui p_k ⟶ *send_query* (k, k, j) *af*

num[k]:=|S|;

wait[k]:=*true*;

un proces blocat p_k la primirea unui mesaj query (i, j, k) executa ::

if mesajul *query* este *de angajare* ⟶

fa p_m din setul S al lui p_k ⟶ *send_query* (i, k, m) *af*

num[i]:=|S|; wait[i]:=*true;*

[] mesajul *query* nu este *de angajare* ⟶

if wait[i] = true ⟶ *send_reply* (i, k, j) la p_j *fi*

fi

un proces blocat p_k, la primirea unui mesaj reply (i, k, j) executa ::

if wait[i]=*true* ⟶

num[i]:=num[i]-1

if num[i] = 0 ⟶

if i = k ⟶ declara interblocare pentru p_k

[] $i \neq k$ ⟶ *send_reply* (i, k, j) la procesul care a trimis mesajul de angajare la p_k

fi

fi

fi

Pentru a garanta că fiecare interblocare va fi detectată, trebuie eliminată restricția ca prelucrarea cu difuzie să poată fi inițiată numai o singură dată de un proces; ca atare, ori de cîte ori se va bloca, un proces va putea iniția o prelucrare cu difuzie. Pentru a se distinge prelucrările cu difuzie inițiate de același proces p_i, mesajele *query* și *reply* vor conține un parametru în plus care să specifice numărul de secvență al prelucrării cu difuzie inițiată de p_i.

BUPT

Capitolul 9.

PROIECTAREA UNUI SUPORT SOFTWARE PENTRU PROGRAMARE DISTRIBUITĂ BAZATĂ PE OBIECTE SI PE TRANZACŢII ATOMICE

Există actualmente un interes crescut în metodologii şi tehnici cu caracter general care simplifică construcţia sistemelor software distribuite robuste şi eficiente. Acest capitol prezintă o astfel de metodologie bazată pe tranzacţii atomice şi include proiectarea unui suport software pentru programarea distribuită bazat pe obiecte (DOSTP). Tranzacţiile atomice micşorează dificultatea scrierii unor programe de aplicaţii distribuite prin simplificarea tratării defectiunilor nodurilor şi prin rezolvarea problemelor de concurenţă.

9.1. Cerinţe de proiectare

Înainte de a prezenta arhitectura sistemului DOSTP se enumeră cerintele de proiectare impuse:

(1) simplitate de apel ale funcţiilor sistemului DOSTP la nivelul aplicaţiilor utilizator;

(2) modularitate şi extensibilitate, în sensul ca pot fi introduşi în sistem noi manageri de obiecte printr-un efort de programare minim; practic, activitatea de programare a unui manager pentru un anumit tip de obiecte se concentrează numai pe structura obiectelor şi a cererilor primite relative la tipul de obiecte respective, pe tratarea cererilor şi rezolvarea problemelor de concurenţă în accesul la obiecte şi pe recuperarea obiectelor în urma defectării nodului pe care acestea au fost rezidente. În ceea ce priveşte conectarea la sistemul tranzacţional al unui manager de obiecte, sistemul DOSTP trebuie să pună la dispoziţie modalităţile şi funcţiile necesare.

(3) sistemul trebuie să ofere posibilitatea execuţiei în paralel a mai multor operaţii asupra unor obiecte diferite;

(4) sistemul trebuie să ofere posibilitatea recuperării obiectelor în urma defecţiunii unuia sau mai multor noduri implicate într-o tranzacţie atomică;

(5) sistemul trebuie să se execute în mod utilizator;

(6) sistemul trebuie să ofere o comunicaţie sigură;

(7) obiectele trebuie să fie portabile (cu modificari minime, absolut necesare) atît în reţele UNIX cat şi în reţele Microsoft Windows (Win32);

(8) tranzacţiile să nu poată fi amînate la infinit. Se cere deci evitarea situaţiei de amînare la infinit sau a restartărilor la infinit.

În urma analizei cerinţelor de mai sus s-a optat ca DOSTP să ofere primitive pentru: (1) conectarea şi deconectarea aplicaţiilor utilizator la sistem; (2) iniţierea unei tranzacţii atomice; (3) execuţia într-o manieră asincronă (returnarea controlului se face imediat după apelul operaţiei) de operaţii asupra unor obiecte, ai căror manageri sunt conectaţi la sistem ; (4) execuţia, într-o manieră sincronă (returnarea controlului se face după execuţia completă a operaţiei) de operaţii asupra unor obiecte, ai căror manageri sunt conectaţi la sistem; (5) aşteptarea terminării execuţiei tuturor operaţiilor asincrone lansate anterior, în cadrul tranzacţiei atomice curente; (6) terminarea normală a unei tranzacţii atomice, deci permanentizarea operaţiilor executate; (7) terminarea anormală a unei tranzacţii atomice.

Se prezintă, în continuare, o descriere în detaliu a primitivelor furnizate de DOSTP:

(1) `int InitDOSTP (char *name)`

Scop:

Permite conectarea unei aplicaţii utilizator la sistemul DOSTP.

Parametri:

name - numele aplicaţiei utilizator

Valoare returnată:

Funcţia returnează SUCCES în cazul în care operaţia a reuşit sau un cod de eroare în caz contrar. Codurile de eroare care pot fi obţinute sunt:

EBADF	-numele aplicaţiei este un şir nul
EDOSTP_INITIALIZE_DONE	-aplicatia era conectata la DOSTP
ECREATE_APL_FIFO	-nu se poate crea canalul de comunicaţie cu aplicaţia

BUPT

(2) **int BeginTransaction (void)**

Scop:
Permite specificarea începerii unei noi tranzacţii atomice.
Valoare returnată:
Funcţia returnează SUCCES în cazul în care operaţia a reuşit sau cod de eroare în caz contrar. Codurile de eroare care pot fi obţinute sunt:

EDOSTP_INITIALIZE_EXPECTED - nu s-a apelat *InitDOSTP()*
EBEGIN_TRANS_DONE -aplicatia are tranzactia deschisa
EOPEN_APL_FIFO -nu se poate deschide canalul de comunicaţie cu aplicaţia
EOPEN_TM_FIFO -nu se poate deschide canalul de comunicaţie cu managerul de tranzacţii
EWRITE_TM_FIFO_FAIL -nu se poate comunică cu managerul de tranzacţii
EREAD_APL_FIFO_FAIL -nu se poate citi canalul aplicaţiei
ETM_CREATE_SHM - man. de tranzactii nu a reusit crearea zonei de memorie partajate
EAPL_ACCESS_SHM_FAIL -nu se poate realiza ataşarea la zona de memorie partajată
ETM_CREATE_SEM - man. de tranzactii nu a reusit crerea semafoarelor
ETM_OPEN_TM_FIFO - man. de tranzactii nu a reusit deschiderea canalul de comunicaţie
ECS_TIME_OUT - eroare time-out la comunicatia cu serverul de comunicatii

(3) int **SyncRequest(char** **objectName,* **void** **sourceAddr,* **void** **destAddr,* **int** *timeOut)*

Scop:
Permite specificarea execuţiei unei operaţii sincrone asupra unui obiect *objectName*. Aplicaţia este pusă în aşteptare pînă în momentul terminării execuţiei operaţiei. Utilizatorul trebuie să rezerve o zonă de memorie în care să transmită parametri operaţiei, a cărei adresă o transmite ca parametru şi o altă zonă de memorie în care sistemul returnează răspunsul la cererea sincronă, a cărei adresă se transmite de asemenea ca parametru. Structura celor două zone de memorie depinde de tipul obiectului. Ea trebuie cunoscută numai de managerul obiectelor de acest tip şi de aplicaţie. Pentru sistemul tranzacţional DOSTP structura acestor zone de memorie nu are nici o importanţă: atît cererea cît şi răspunsul sunt tratate ca un şir de octeti.
Parametri:
sourceAddr - adresa de început a zonei de memorie la care apelantul a depus cererea de execuţie a operaţiei sincrone
destAddr - adresa de început a zonei de memorie la care sistemul returnează răspunsul la cererea sincronă
objectName - numele obiectului implicat în tranzacţie
timeOut - valoarea maximă pentru timpul de aşteptare al răspunsului
Valoare returnată:
Funcţia returnează valoarea SUCCES în cazul în care cererea a fost tratată de managerul de obiecte, ceea ce nu inseamnă că operaţia s-a executat cu succes, sau un cod de eroare în caz contrar. Codurile returnate pot avea urmatoarele valori:

EBADF -numele obiectului este un şir nul
EDOSTP_INITIALIZE_EXPECTED - nu s-a apelat *InitDOSTP()*
EINVALID_TRANS -nu s-a executat *BeginTransaction()*.
ENO_BUF -nu mai există loc în buferele sistemului pentru aceasta cerere şi pentru răspuns
ETIME_OUT -s-a depaşit timpul maxim de aşteptare specificat în parametrul de intrare
ETOO_MANY_OBJECTS - s-a depaşit numărul maxim de obiecte permise într-o tranzacţie
EOBJ_NOT_FOUND - obiect inexistent

(4) **int AsyncRequest (char** **nameObeject,* **void** **sourceAddr,* **void***destAddr,* **int** **pErrorCode,* **int** *timeOut)*

Scop:
Permite specificarea execuţiei unei operaţii asupra unui obiect *objectName*. Controlul execuţiei este redat aplicaţiei imediat după apelul funcţiei. Ca şi în cazul funcţiei precedente, SyncRequest, utilizatorul trebuie să rezerve două zone de memorie pentru cererea asincronă, respectiv pentru răspuns, ale căror adrese de început se dau ca parametri de intrare. Formatul acestor zone trebuie cunoscut numai de managerul de obiecte. Avînd în vedere că răspunsul la cerere se obţine la *destAddr* după ce controlul a fost returnat aplicaţiei, la *pErrorCode* se livrează o prima indicaţie despre desfaşurarea execuţiei, în sensul ca cererea a fost sau nu acceptată de sistem.
Parametri:
nameObject -numele obiectului

BUPT

sourceAddr -adresa de început a zonei de memorie la care apelantul a depus cererea de execuție a operației sincrone
destAddr -adresa de început a zonei de memorie la care sistemul returnează răspunsul la cererea asincrona
pErrorCode -adresa la care sistemul va returna codul de eroare obținut în urma tratării cererii de catre sistemul tranzacțional. Codul erorii obținute în urma tratării cererii de catre managerul de obiecte se codifică în răspunsul livrat la *destAddr*.
timeOut -aloarea maxima pentru timpul de așteptare al răspunsului. Daca aplicația nu primește răspuns în acest interval de timp la adresa *pErrorCode* se va returna ETIME_OUT.

Valoare returnată:
Funcția returnează SUCCES în cazul în care cererea a fost acceptată de sistem sau unul dintre urmatoarele coduri de eroare:
EBADF -numele obiectului este un șir nul
EDOSTP_INITIALIZE_EXPECTED -nu s-a apelat *InitDOSTP ()*
EINVALID_TRANS -nu s-a apelat *BeginTransaction()*
ENO_BUF -nu mai există loc în buferele sistemului pentru această cerere și pentru răspuns
La *pErrorCode* se pot returna urmatoarele valori:
ETIME_OUT -s-a depașit timpul maxim de așteptare specificat în parametrul de intrare.
ETOO_MANY_OBJECTS - s-a depașit numărul maxim de obiecte permise într-o tranzacție
EOBJ_NOT_FOUND - obiect inexistent

(5) **`int Syncronize (void)`**

Scop:
Permite așteptarea terminării tuturor operațiilor asincrone lansate anterior.
Valoare returnată:
Funcția returnează SUCCES sau următorul cod de eroare atunci cînd nu s-a inițiat nici o tranzacție dar se cere execuția ei:
EDOSTP_INITIALIZE_EXPECTED - nu s-a apelat *InitDOSTP ()*
EINVALID_TRANS -nu s-a executat *BeginTransaction()*.

(6) **`int EndTransaction (void)`**

Scop:
Permite terminarea normală a unei tranzacții atomice lansate anterior. Funcția comandă sistemului DOSTP execuția celor două faze ale procedurii *two-phase commit*. Înainte de a se încerca desavârșirea tranzacției aplicația trebuie să încerce mai întîi terminarea tuturor operațiilor asincrone lansate, printr-un apel *Synchronize()*.
Valoare returnată:
Funcția returnează SUCCES în cazul în care desavîrșirea tranzacției a reușit sau următoarele coduri de eroare:
EDOSTP_INITIALIZE_EXPECTED - nu s-a apelat *InitDOSTP ()*
EINVALID_TRANS - nu s-a apelat *BeginTransaction()*
ESYNCRONIZE_EXPECTED - trebuie apelată funcția *Synchronize()* și apoi *EndTransaction()*
ETIME_OUT - desavîrșirea tranzacției nu a reușit în intervalul de timp maxim
ECOMMIT_FAIL - eroare în desavârșirea tranzacției (obiectele nu au putut realiza în totalitate operațiile cerute)

(7) **`int AbortTransaction (void)`**

Scop:
Permite terminarea anormală sau anularea unei tranzacții atomice lansate anterior. Funcția trebuie apelată obligatoriu după orice operație asupra unui obiect nereușită.
Valoare returnată:
Funcția returnează SUCCESS sau valorile:
EDOSTP_INITIALIZE_EXPECTED -nu s-a apelat *InitDOSTP ()*
EINVALID_TRANS -nu s-a apelat *BeginTransaction()*

(8) **`int CloseDOSTP (void)`**

Scop:
Funcția se apelează pentru a deconecta aplicația de la sistemul TCP.
Valoare returnată:
Funcția returnează SUCCESS sau valorile:
EDOSTP_INITIALIZE_EXPECTED -nu s-a apelat *InitDOSTP ()*

BUPT

9.2. Modelul sistemelor DOSTP

În contextul celor prezentate în capitolele 6, 7 şi 8 se apreciază că:

(1) Modelul de arhitectură al sistemului DOSTP este *modelul integrat* (vezi 7.2.2). În cadrul acestui model toate nodurile (în care se iniţiază tranzacţii) sunt dotate cu manageri de tranzacţii şi cu manageri de obiecte, daca există obiecte pe acele noduri.

(2) Pentru realizarea tranzacţiilor distribuite se foloseşte modelul în care se crează cîte un *proces agent* în fiecare nod implicat în execuţia unei tranzacţii (vezi 7.2.2); procesul agent va obţine accesul la obiectul de pe acel nod şi va realiza dialogul manager de tranzacţie - manager de obiect.

(3) DOSTP utilizează *varianta dinamică a obiectului activ* (vezi 6.3) în care, atunci când se recepţionează o cerere de implicare a unui obiect într-o tranzacţie, se crează câte un proces server pentru fiecare obiect. Toate operaţiile cerute de acea tranzacţie pentru obiectul respectiv vor fi tratate numai de acel proces server. Avantajul acestei metode este faptul că nu limitează numarul de tranzacţii ce pot fi tratate simultan la nivelul unui obiect (decât, evident, de numărul maxim de procese concurente permise de sistemul de operare, ceea ce, în mod normal, este puţin probabil de atins).

(4) Localizarea obiectelor se face folosind soluţia *serverelor de nume distribuite* (vezi 6.4.1). Sistemul crează câte un server de nume pe fiecare nod. Există un manager central, care cunoaşte toate nodurile componente ale sistemului DOSTP şi la care se conectează toate nodurile la intrarea în sistem. Acest manager are şi rolul de a actualiza informaţiile din serverele de nume, la conectarea sau eliminarea unui manager de obiecte în sistem, prin transmisia de mesaje către toate serverele de nume.

(5) Operaţiile asupra obiectelor sunt apelate folosind *transmisia de mesaje* (vezi 6.4.2), strategie tipică pentru sistemele în care se implementează modelul obiectului activ. Când o aplicaţie apelează o operaţie asupra unui obiect, parametrii cererii sunt împachetaţi într-un mesaj; mesajul este transmis apoi procesului manager al obiectului, după ce obiectul a fost localizat folosind serverul de nume. Procesul manager al obiectului receptionează mesajul, despachetează parametrii din mesaj, executa operaţia şi împachetează rezultatul într-un alt mesaj pe care-l transmite apelantului. Pentru apelurile la obiecte situate în acelaşi nod cu aplicaţia care a iniţiat tranzacţia, în DOSTP s-a optat ca mesajele să fie transmise via o zona de memorie comună, partajată de aplicaţie şi de managerul de obiecte.

(6) In DOSTP se consideră că se lucrează cu cereri de tratare a unor operaţii a căror execuţie nu depăşeşte un timp prestabilit, astfel încît răspunsul la o cerere se consideră şi o confirmare a recepţionării cererii (vezi 3.1.3).

(7) În ceea ce priveşte gestionarea operaţiilor asupra obiectelor (vezi 6.3.1) DOSTP foloseşte *mecanismul tranzacţiilor atomice*. Cînd toate operaţiile grupate în cadrul unei tranzacţii au fost executate cu succes, sistemul este cel care realizează procedura de permanentizare a modificarilor. Dacă cel puţin o operaţie se termina prin eroare, tranzacţia trebuie terminată prin abort.

(8) Protocolul ales pentru terminarea tranzacţiilor distribuite este protocolul *two-phase commit* (vezi 7.3.1). Protocolul presupune existenţa unui coordonator al comiterii tranzacţiei; coordonatorul unei tranzacţii distribuite este întotdeauna în DOSTP managerul de tranzacţii de pe nodul de la care s-a iniţiat tranzacţia respectivă. Coordonatorul tranzacţiei supervizează comunicaţia cu toti agenţii de la distanţă ai unei tranzacţii.

(9) Schema de sincronizare folosită în DOSTP pentru sincronizarea operaţiilor la nivelul aceluiaşi obiect, în cadrul unui managerde obiecte, se poate încadra în *schemele optimiste* prezentate pe scurt în 6.3.2 şi mai în detaliu în 7.4.6. Schema de sincronizare folosită este proprie managerului de obiecte. Pot fi realizaţi mai mulţi manageri de obiecte pentru acelaşi tip de obiecte, fiecare folosind propria sa schemă de sincronizare, de o maniera optimistă sau pesimistă. Pentru managerii de date realizaţi pînă în momentul de faţă s-a folosit o schema de sincronizare optimistă. Se apreciază că o direcţie de extindere a acestui sistem poate consta în testarea mai multor scheme de sincronizare pentru acelaşi tip de obiect. Se poate utiliza chiar şi o schemă hibridă care funcţionează astfel: monitorizîndu-se frecvenţa de conflicte ale cererilor de operaţii asupra unui obiect, atunci cînd frecvenţa este mai mică decît limita maximă, se utilizează varianta optimistă, iar cînd conflictele sunt mai dese (frecvenţa mare) se utilizează varianta pesimistă. Schemele de sincronizare optimiste permit cel mai mare grad de concurenţă posibil la nivelul unui obiect; operaţiile nu sunt puse niciodată în aşteptare. Preţul plătit pentru această performanţă este faptul ca acţiuni executate fără erori pot fi terminate anormal (la *commit*) deoarece la desavîrşirea tranzacţiilor se testează condiţiile de seriabilitate astfel încît informatiile de stare luate în considerare de o operaţie să nu intre în conflict cu modificările realizate de orice altă operaţie care s-a terminat (a fost acceptată în faza *prepare-to-commit*); în plus, trebuie menţinute în memorie mai multe copii ale obiectului.

(10) In ceea ce priveşte reprezentarea obiectelor in memorie (vezi 6.5.) se precizează faptul ca DOSTP lucrează cu o copie persistentă a obiectului, memorată într-un fişier care, la încărcarea obiectului, este mapat in memorie. Fiecare manager de obiect menţine, local unei tranzacţii, toate modificările făcute asupra obiectului de către acea tranzacţie care se permanentizeaza in faza de *commit*.

BUPT

(11) Pentru siguranța obiectelor, DOSTP folosește *metoda recuperării obiectelor*, varianta *roll-back* (vezi 6.3.3 și 7.3.4). Se lucrează cu un sistem de recuperare al obiectelor bazat pe *fisiere jurnal (log)* care folosește protocolul *Write-Ahead Logging* (vezi 7.3.4.1) și tehnicile *Redo-only-logging* și *Value-logging* (vezi 7.3.4.2 și 7.3.4.3). Tehnica *Redo-only-logging* este avantajoasă ținînd cont că se folosește o schemă de sincronizare optimistă; actualizările sunt întîrziate pînă în momentul comiterii tranzacției, astfel încît informațiile de *undo* nu trebuie scrise în log. Sistemul garantează că, odată ce tranzacția execută *commit*, orice actualizări întîrziate sunt realizate, chiar dacă intervine un incident hardware. Tehnica constă în crearea unor *liste de intentii* care să cuprindă actualizările în așteptare și de a forța scrierea acestor liste în log, înainte de faza *prepare-to-commit*..

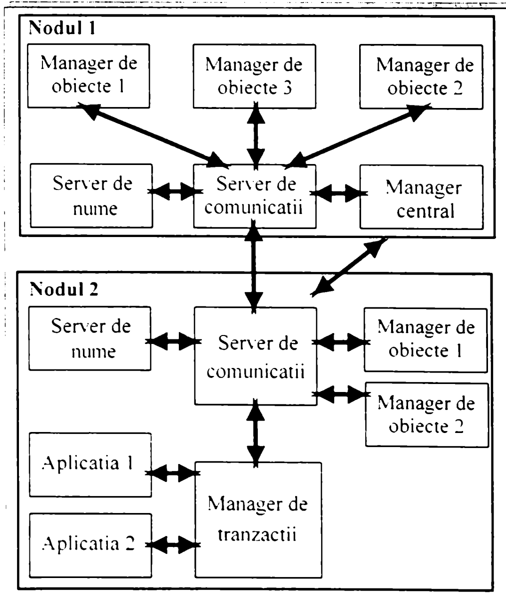
În ceea ce privește structura sistemului de recuperare (vezi 7.3.4.4), elementele sale sunt încorporate la nivelul managerilor de obiecte și la nivelul managerilor de tranzacții.

(12) În ceea ce privește controlul interblocării obiectelor se fac două observatii (vezi 7.4.6.1): (1) Deoarece se folosește o schemă de sincronizare optimistă nu pot apare situații de interblocari și restartări ciclice. (2) Pot apare însă situații de restartare infinită. Prin urmare, pentru respectarea cerintei 8 de proiectare impusă sistemului DOSTP, care interzice apariția restartărilor la infinit, s-a optat pentru implementarea unui mecanism *Wound-die* (vezi 8.2). Acest mecanism evită restartările la infinit ale tranzacțiilor, dar impune asocierea unor mărci de timp tranzacțiilor și reluarea tranzacțiilor după abortare cu aceeași marcă de timp.

Deoarece tranzacțiile pot fi inițiate de pe orice nod din sistem, pentru alocarea unei mărci de timp unice unei tranzacții, care de fapt este o prioritate asociată unei tranzacții- cu cît prioritatea are o valoare mai mică, tranzacția este mai prioritară, sau, altfel spus, are o marcă de timp mai mică, deci este mai veche în sistem și deci mai prioritară)- pot fi folosite două soluții: o soluție complet distribuită care presupune însă circulație de mesaje pentru alegerea mărcii de timp (sau priorității) în regim de excludere reciprocă (poate fi folosit unul din algoritmii prezentați la 4.3.2, 4.3.3, 4.3.4) sau o soluție cu proces coordonator. În DOSTP s-a optat, pentru a simplifica circulația de mesaje, pentru soluția unui proces coordonator. În DOSTP, pe un nod fixat există un *manager central* cu dublu rol: (1) primul rol a fost prezentat la punctul (2); el este legat de conectarea nodurilor la sistemul DOSTP și de conectarea sau eliminarea unor manageri de obiecte; (2) al doilea rol al managerului central constă în alocarea de priorități tranzacțiilor, la inițierea lor. La recepționarea unei cereri *BeginTransaction()* fiecare manager de tranzacție contactează managerul central pentru a-i livra prioritatea tranzacției. Cînd o tranzacție este reluată, prioritatea tranzacției nu se modifică; astfel, după cîteva încercări nereușite, devine cea mai veche din sistem și poate fi terminată normal.

9.3. Arhitectura generală a sistemului DOSTP

Componentele sistemului DOSTP dintr-un nod în care se inițiaza tranzacții și în care există și manageri de obiecte și dintr-un nod în care există numai obiecte sunt prezentate în figura 9.3.1. Managerul central se poate găsi pe orice nod al sistemului, însa fixat și cunoscut. Dacă într-un nod nu se implementează obiecte, atunci managerii de obiecte pot lipsi; pe de alta parte, într-un nod pot fi rezidenți mai mulți manageri de obiecte, pentru același tip de obiecte sau pentru tipuri diferite. Analog, managerul de tranzacții poate lipsi dintr-un nod, dacă de pe acel nod nu se intenționează să se execute aplicații care folosesc sistemul DOSTP. Pe un nod există (eventual) un singur manager al tranzacțiilor, un singur server de nume, un singur server pentru comunicatii, dar pot exista mai mulți manageri de obiecte și mai multe aplicații.



Pe scurt, funcțiile componentelor DOSTP sunt:

1. *Managerii de obiecte* stochează informațiile de stare ale obiectelor, realizează mecanismul de sincronizare specific tipului implementat și execută operațiile asupra obiectelor. Ei cooperează cu managerii de recuperare pentru a implementa protocolul pentru recuperarea tranzacțiilor. Managerii de obiecte sunt proiectați folosind *tehnica programării orientate pe obiecte* astfel încît integrarea unui manager de obiecte în sistemul tranzacțional să

BUPT

nu depindă de tipul obiectului. Managerii de obiecte sunt singurele componente care cunosc structura cererilor aplicațiilor și structura obiectelor.

2. *Serverele de nume* (există cîte un server în fiecare nod conectat la DOSTP) realizează localizarea obiectelor în rețea; ele mențin structuri de date care memorează poziția obiectelor pe noduri.

3. *Serverele de comunicație* transmit mesajele inter-noduri, generează procesele agent în cadrul tranzacțiilor distribuite, și detectează defecțiunile altor noduri cu care inițiază comunicația.

4. *Managerii de tranzacții* coordonează inițierea, desăvîrșirea și terminarea anormală a unei tranzacții.

5. *Sistemul de recuperare* are trei funcții majore: (a) înainte de transmisia confirmării mesajului *prepare-to-commit*, înscrie listele de intenții în fisierul log; formatul înregistrarilor depinde de tipul obiectelor, de aceea s-a considerat oportun ca această funcție să fie înglobată în managerii de obiecte; (b) o altă funcție este recuperarea obiectelor în urma defectării nodului respectiv și apoi a repunerii în funcțiune a lui; și această funcție depinde de structura obiectelor și a fost înglobată în managerii de obiecte; (c) a treia funcție constă în luarea deciziei de a executa *commit* sau *abort* în urma analizei fișierelor de log după o defecțiune a unui nod coordonator al unei tranzacții urmate de repunerea lui în funcțiune. Aceste fișiere de log care conțin informații despre starea tranzaciilor și obiectelor implicate în tranzacții se mențin la nivelul managerului de tranzacții, în calitatea sa de coordonator al tranzacțiilor distribuite, și, în consecință, funcția (c) este înglobată la nivelul managerului de tranzacții.

7. *Procesele de aplicație* inițiază tranzacțiile și apelează operațiile implementate în serverele de date.

8. *Managerul central* este un singur proces în tot sistemul, pe un nod fixat. El are două funcții: (a) ține evidența nodurilor conectate la sistem și transmite serverelor de nume, la conectarea lor, localizarea obiectelor din sistem; (b) livrează prioritățile pentru tranzacții.

9.3.1. Procesele utilizator ale sistemului DOSTP

Procesele utilizator (sau de aplicație) ale sistemului DOSTP constituie procesele de nivelul cel mai înalt în utilizarea tranzacțiilor, procesele care inițiază tranzacțiile. Aplicațiile apelează operații asupra obiectelor prin trimiterea de mesaje și recepționarea rezultatelor de la managerii de date.

Spre deosebire de managerii de date, aplicațiile nu memorează obiecte în corpul lor și de aceea nu acceptă mesaje de la alte servere de date. Aplicațiile nu sînt afectate de mecanismele de recuperare și deci nu sunt restartate după o defectiune de nod.

Pentru a putea apela operații asupra obiectelor, aplicațiile trebuie să cunoască structura cererilor recunoscute de manageri pentru acele tipuri de obiecte. Pe de alta parte, aplicațiile trebuie să realizeze conectarea cu sistemul DOSTP. Aceste cerinte sunt îndeplinite prin linkeditarea aplicațiilor cu un set de funcții puse la dispoziție aplicațiilor de sistemul DOSTP. Pe un nod al sistemului în care se execută aplicații, trebuie să existe un manager de tranzacții; acesta poate deservi un numar practic nelimitat de aplicații locale.

9.3.2. Managerul tranzacțiilor

Managerul tranzacțiilor este un proces a carui funcție este de a coordona inițierea și terminarea unei tranzacții. Informațiile despre starea tranzacțiilor sunt furnizate prin două clase de mesaje recepționate de managerul tranzacțiilor de la alte componente STAD.

Prima clasă de mesaje declanșează inițierea și terminarea unei tranzacții. A două clasă informează managerul de tranzacții ce noduri la distanță și ce manageri locali sînt participanți într-o tranzacție dată.

Procesul care a inițiat o tranzacție trebuie să fie obligatoriu și cel care încearcă să o termine cu *commit*. Pentru terminarea unei tranzacții, managerul tranzacțiilor utilizează o variantă de protocol *two-phase commit*. Protocolul este bazat pe construcția unui arbore; fiecare nod este coordonator de tranzacție pentru nodurile sale fii. Un nod A este părintele unui alt nod B dacă și numai dacă nodul A a fost primul nod care a apelat o operație în nodul B în decursul aceleiasi tranzacții. Informațiile referitoare la relațiile unui nod cu alte noduri ascendente sau descendente direct în arborele tranzacțiilor sînt păstrate în managerul de tranzacții.

9.3.3. Serverele de comunicații

Serverele de comunicații sunt procese rezidente în fiecare nod, care acționează ca un tampon între emițătorul unui mesaj și receptorul sau, realizînd localizarea transparentă a proceselor. Perechea de procese corespondente, servere ale comunicațiilor între cele două noduri asigură astfel gestiunea comunicațiilor.

Serverele de comunicație furnizează și funcția de monitorizare a rețelei. Un server de comunicație detectează defecțiunile de comunicație și de noduri în urmatoarele situații:

1. *pierderea conexiunii*. Indică o defecțiune a nodului cu care se comunică în mod direct. Este detectată de managerul comunicațiilor, fie datorită lipsei mesajului de confirmare, fie datorită nerecunoașterii unei conexiuni deja folosite și confirmate anterior.

2. *erori de tip time-out* . Serverul comunicațiilor furnizează proceselor emițătoare (din nodul său) un port pentru a fi adresat atunci cînd procesul receptor este la distanță. El nu cunoaște noțiunea de tranzacție dar

BUPT

este responsabil de construcția arborelui tranzacției; înregistrarea identității nodurilor fii este menținută însă la nivelul managerului de tranzacții.

Serverele de comunicații sunt singurele procese dintr-un nod DOSTP care comunică cu alte noduri. Un server de comunicație dintr-un nod este contactat de managerul de tranzacții, de serverul de nume și de managerii de obiecte pentru orice schimb de mesaje cu alte procese de pe alte noduri.

9.3.4. Serverele de nume

Sistemul DOSTP prevede un serviciu de gestiune a numelor permanente pentru obiecte. Procesele servere de nume se execută în fiecare nod și cooperează cu serverul central pentru gestiunea numelor permanente. Serverele de nume livrează managerilor de comunicații locali, informații despre localizarea obiectelor din rețea, în momentul în care managerii de tranzacție primesc o cerere de operație asupra unui obiect nou implicat într-o tranzacție.

9.3.5. Managerul central

Managerul central este un proces unic în sistemul DOSTP. El este contactat în trei situații:

(1) la conectarea unui nod în sistem, la începutul execuției serverului de comunicații. Serverul de comunicații își anunță prezența în sistem, după care primește de la managerul central o listă cu toate obiectele din sistem pe care o transmite serverului de nume din același nod.

(2) la introducerea sau eliminarea unui nou manager de obiecte în sistem;

(3) la inițierea unei tranzacții cînd managerul de tranzacții, prin intermediul serverului de comunicații cere o prioritate pentru noua tranzacție.

Pentru acordarea identificatorilor si prioritatilor tranzactiilor se foloseste un fisier local care retine valoarea urmatoare de acordat identificatorului, respectiv prioritatii unei tranzactii.

9.3.6. Managerii de obiecte

Managerii de date încapsulează două tipuri de date, obiectele și informația de sincronizare asociată și patru tipuri de funcții:

(1) funcțiile care implementează operațiile care actionează asupra obiectelor;

(2) funcțiile care implementează mecanismul pentru sincronizarea tranzacțiilor în accesul la obiecte;

(3) funcțiile care implementează dialogul cu managerul de comunicații;

(4) funcțiile care implementează restaurarea obiectelor după o terminare anormală sau defect al unui nod.

Funcțiile din categoriile (1), (2) și (4) depind de tipul obiectului gestionat de manager; cele din categoria (3) nu depind de tipul obiectului. Avînd în vedere aceasta observație o consecință este imediată: folosind tehnici de programare specifice programării orientate pe obiecte trebuie să se separe în clase separate funcțiile de tipul (3) și funcțiile de tipurile (1), (2) și (4); în acest fel, pentru proiectarea unor noi manageri de obiecte, vor fi rescrise numai funcțiile de tipul (1), (2) și (4).

Fiecare manager de obiecte acceptă mesaje care invocă execuția unor operații și returnează rezultatul execuției în alte mesaje.

După un defect al unui nod, se startează noi procese manageri de date pentru a accesa obiectele protejate. În aceste situații, starea obiectelor partajabile se restaurează; la restaurare, contribuie funcțiile din categoria (4).

Managerii de date conțin și implementarea mecanismului ales pentru sincronizarea tranzacțiilor în DOSTP; acesta poate folosi o schema pesimistă, optimistă sau o schema hibridă. Managerii de obiecte vor conține relațiile de compatibilitate pentru lock-uri, tabelele care descriu care lock-uri sunt deținute de tranzacții și identitatea acestor tranzacții și codul care manipulează aceste structuri de date.

Se apreciază că managerii de obiecte constituie o abstractizare a unor tipuri de date puse la dispoziția aplicațiilor.

9.3.7. Sistemul de recuperare

Sistemul de recuperare nu este implementat în DOSTP ca proces distinct; el se compune din module plasate în managerii de obiecte și în managerii de de tranzacții precum și din fișiere jurnal (sau fișiere de log) plasate analog lîngă managerul de tranzacție aferent sau lîngă managerul de obiecte.

Sistemul de recuperare se constituie din:

(1) codul procedurilor pentru recuperarea obiectelor, proceduri lansate în execuție după defectarea și apoi repunerea în funcțiune a unui nod și plasate în managerii de obiecte;

(2) codul funcțiilor pentru scrierea articolelor în fișierele log, conform tehnicilor *Redo-only-logging* și *Value-logging* plasate în managerii de obiecte;

(3) codul funcțiilor pentru analiza fișierelor log în scopul determinării stării tranzacțiilor în momentul apariției defecțiunii unui nod, plasat în managerii de tranzacție.

BUPT

(4) fişierele de log care reţin informaţii despre tranzacţiile active şi despre obiectele implicate în aceste tranzacţii; aceste fişiere sunt plasate lîngă managerul de tranzacţie, coordonator al tranzacţiei respective.

(5) fişierele de log care menţin informaţiile cu care trebuie actualizată copia permanentă a obiectelor în momentul execuţiei fazei *commit*.

9.4. Proiectarea elementelor componente ale sistemului DOSTP

În subcapitolul de faţă se prezintă modul de proiectare al elementelor componente ale sistemului DOSTP, insistîndu-se asupra structurilor de date utilizate şi a circulaţiei mesajelor între componente.

9.4.1. Structuri de date ale sistemului

Înainte de a se prezenta structurile de date ale sistemului este necesar să se arate pe scurt modul de desfăşurare al unei tranzacţii în cadrul sistemului DOSTP şi modul de comunicare între elementele sistemului DOSTP.

Există patru momente distincte în funcţionarea sistemului DOSTP:

(M1) Momentul conectării unui nod la sistem, cînd se lansează în execuţie serverul de comunicaţii şi serverul de nume. Daca pe acel nod urmează să se execute aplicaţii se lansează în execuţie şi managerul de tranzacţii. Serverul de comunicaţii contactează managerul central şi primeşte o listă cu localizarea tuturor obiectelor introduse în sistem. Din acest moment, aplicaţiile pot să lucreze cu tranzacţii.

(M2) Momentul introducerii sau eliminării unui obiect din sistem, cînd managerul de obiecte trimite un mesaj adecvat managerului central care anunţă toate nodurile din sistem (via servere de comunicaţii - servere de nume) despre modificarea operată în lista de obiecte.

(M3) Desfaşurarea unei tranzacţii are loc astfel:

(3.1) Aplicaţia lansează un apel *BeginTransaction()*, a cărui urmare este transmiterea unui mesaj ALP-BEGIN-TRANSACTION la managerul de tranzacţii.

(3.2) Managerul de tranzacţii (TM) care asteaptă în buclă continuă receptionarea de mesaje pe canalul *chan-TM*, primeşte mesajul şi crează un proces fiu ce crează o zona de memorie partajată şi un set de semafoare pentru accesul la aceasta zonă şi trimite identificatorul zonei şi semafoarelor catre TM; acesta va comunica cu aplicaţia şi cu serverul de comunicaţii *numai prin zona de memorie partajată* în cadrul tranzacţiei pe care o va gestiona.

(3.3) Procesul fiu transmite un mesaj TM-BEGIN-TRANSACTION către serverul de comunicaţii (CS) care, la rîndul sau, crează un fiu al carui scop va fi tratarea tranzacţiei respective. Procesul fiu va contacta managerul central pentru obţinerea priorităţii tranzacţiei.

(3.4) După obţinerea prioritătii, procesul fiu al CS-ului, confirmă începerea tranzacţiei transmiţînd un mesaj ACK-TM-BEGIN-TRANSACTION fiului TM-ului.

(3.5) Fiul TM-ului confirmă începerea tranzacţiei trimiţînd aplicaţiei un mesaj ACK-TM-BEGIN-TRANSACTION.

(3.6) Toate cererile de efectuare a unor operaţii asupra obiectelor sunt trimise numai prin memoria partajată.

(3.7) Prima cerere dintr-o tranzacţie către un obiect este defalcată la nivelul serverului de comunicaţii în două cereri: o primă cerere transmisă într-un canal creat static de catre obiect (în care asteaptă cereri de implicare) *chan-OB* şi cererea propriu-zisă transmisă în memoria partajată.

(3.8) Pentru obiectele aflate pe alte noduri, se angajează un dialog între serverele de comunicaţie: cel de pe nodul aplicaţiei şi cel de pe nodul obiectului. Este sarcina serverului de comunicaţie de pe nodul obiectului să creeze o zona de memorie partajată cu managerul obiectului cînd se implică obiectul respectiv în tranzacţie.

(3.9) După primirea unei cereri de implicare a unui obiect într-o tranzacţie, managerul obiectului crează un proces fiu care va trata toate cererile pentru acel obiect provenind din tranzacţia respectivă.

(3.10) La încheierea tranzacţiei, aplicaţia transmite (tot prin memoria partajată) mesajul END-TRANSACTION sau ABORT-TRANSACTION.

(3.11) Procesul fiu creat de TM pentru supervizarea tranzacţiei este cel care asigură înfaptuirea celor două faze ale protocolului *two-phase commit*.

(3.12) După terminarea tranzacţiei, procesele fii create de TM, CS, OBM se distrug.

(M4) După defectarea unui nod şi repunerii lui în funcţiune trebuie luate în considerare două situaţii:

(4.1) nodul în cauză este cel pe care se afla coordonatorul de tranzacţie

(4.2) nodul în cauză este un nod pe care se aflau numai obiecte.

După prezentarea sumară de mai sus a modului de comunicaţie între componentele sistemului DOSTP în cele patru momente distincte din funcţionare, rezulta cîteva concluzii în ceea ce priveşte structurile de date ale sistemului:

(1) procesele componente ale sistemului vor avea două tipuri de canale de comunicaţie: (A) *canale statice*, fixate din momentul compilarii, pe care se transmit mesajele de la iniţierea unei tranzacţii (3.1, 3.2, 3.3), implicarea unui nou obiect într-o tranzacţie (3.7, 3.8), conectarea unui nod la sistem (M1), inserarea sau

BUPT

eliminarea unui obiect din sistem (M2) și (B) *canale dinamice*, create în timpul execuției, în momentul cînd procesele fii ale serverelor de comunicație de pe noduri inițiază dialogul în cadrul unei tranzacții.

(2) într-un nod, pe timpul desfașurării unei tranzacții, mesajele se transmit printr-o zonă de memorie partajată la care accesul se face controlat de semafoare, după modelul *producator-consumator*. Astfel, dacă, de exemplu, un proces fiu al lui TM care supervizează tranzacția T1 (pe scurt, TM-T1) trebuie să transmită un mesaj procesului fiu al lui CS, care supervizează T1 (CS-T1), atunci el execută o operație Up() asupra semaforului asociat lui CS-T1, deblocînd procesul CS-T1 care, eventual, astepta la semafor. În cele ce urmează, se prezintă structurile de date globale ale sistemului:

```
enum   kind = (APL_BEGIN_TRANSACTION, END_TRANSACTION, REQUEST, FIRST_REQUEST,
       TRANSACTION_ACCEPTED, TRANSACTION_ENDED, ACK_APL_BEGIN_TRANSACTION,
       CS_END, APL_END, ABORT, COMMIT, ACK_ABORT,ACK_COMMIT,CONNECT, DISCONNECT,
       RECORD, DELETE, GET_TRANS_ID, GET_OBJECT, ACK_TM_BEGIN_TRANSACTION,
       TM_BEGIN_TRANSACTION, TM_END, BEGIN_TRANSACTION, TRANSACTION_INTERRUPTED,
       INTERRUPT,TRANSACTION_NOT_ACCEPTED, TRANS_ID_ACK, ACK_RECORD, OBJ_FOUND,
       UNKNOWN, ACK_REQUEST, ACK_FIRST_REQUEST, RECORD_END)
enum   errors = (EBADF , EDOSTP_INITIALIZE_DONE, ECREATE_APL_FIFO,
       EDOSTP_INITIALIZE_EXPECTED , EBEGIN_TRANS_DONE , EOPEN_APL_FIFO
       EAPL_OPEN_TM_FIFO , EWRITE_TM_FIFO_FAIL, EREAD_APL_FIFO_FAIL,
       EAPL_ACCESS_SHM_FAIL, EINVALID_TRANS , ENO_BUF, ETIME_OUT , NETW_ERROR ,
       ESYNCRONIZE_EXPECTED, ECOMMIT_FAIL, ETM_CREATE_SHM, ETM_ACCESS_SHM_FAIL,
       ETM_CREATE_SEM, ETM_OPEN_TM_FIFO, ETOO_MANY_OBJECTS, EOBJ_NOT_FOUND)

const  MAX_SLOTS = ...           # numarul maxim de pachete din zona de memorie partajata
const  MAX_OBJ_LEN = ...         # dimensiunea maxima a numelui unui obiect
const  MAX_APL_LEN = ...         # dimensiunea maxima a numelui unei aplicatii
const  S_FIRST_UNUSED = 6
const  SEM_NO = ...              # numarul de semafoare alocate obiectelor dintr-o tranzactie
const  MAX_TRY=...               # nr. max. de incercari pentru implicarea unui nod la distanta
const  MAX_OBJ_NO = SEM_NO - S_FIRST_UNUSED   # nr maxim de obiecte dintr-o tranzactie
const  TIME_OUT_APL_TM_BT = ...               # valori de time-out
const  TIME_OUT_APL_TM_SREQ = ...
const  TIME_OUT_APL_TM_AREQ = ...
const  TIME_OUT_TM_CS_BTACK = ...
const  TIME_OUT_REQ =...
const  TIME_OUT_RES =...

type SHMEM = record of                     # structura zonei de memorie partajata
 (sbtTable[1 : MAX-SLOTS]: PACKET,         # zona de pachete
  freePackList[1:MAX_SLOTS]: int,          # lista de pachete libere si ocupate
  freePackListIdx: int,                    # index în lista de pachete; indica, la dreapta pachete
                                           # libere, iar la stinga pachete ocupate
  aplList [1:MAX_SLOTS]: int,              # coada de pachete trimise opera]iei
  aplListidx: int,                         # index în coada de pachete trimise aplicatiei;
  transmanList [1:MAX_SLOTS]: int,         # coada de pachete trimise managerului de tranzactii (fiu)
  transManListidx: int,                    # indice în coada de pachete de mai sus
  commServList [1:MAX_SLOTS]: int,         # coada de pachete trimise serverului de comunicatii fiu
  commServListidx : int,                   # indice în coada de pachete de mai sus
  objectList [1:MAX_OBJ][1:MAX_SLOTS]: int,  # cozile de pachete trimise managerului de obiecte
  objectListidx : int                      #indice în cozile de mai sus
)
type PACKET = record of          # structura unui pachet din zona de memorie partajata
 (type : int,                    # tipul pachetului
  errorCode : int,               # codul erorii
  transId : int,                 # identificatorul tranzactiei în care este implicat obiectul
  priority : int,                # prioritatea tranzactiei
  seqNo : int,                   # numar de utilizare al slotului
  aplSlot : int,                 # numarul slotului utilizat de aplicatie
  aplSlotUsage : int,            # numarul slotului aplicatiei
  objId : int,                   # utilizat intern de TM pentru a identifica obiectele
```


BUPT

```
  objName [1:MAX_OBJ_LEN]: char,        # numele unui obiect
  text : REQUEST                        # cererea propriu-zisa, structura ei nu are importanta decit
                                        #pentru aplicatie si managerul de obiecte
)
type BT_PACKET  = record of             # structura unui pachet transmis la initierea unei transactii
 (type: int,                            # tipul mesajului
  transId: int,                         # identificatorul tranzactiei
  pri: int,                             # prioritatea tranzactiei
  aplName[1:MAX_APL_LEN]: char          # numele aplicatiei
)
type BT_ACK_PACKET = record of          # structura unui pachet transmis la initierea unei tranzactii
 (type: int,                            # tipul mesajului
  errCode: int,                         # codul de eroare
  transId: int,                         # identificatorul tranzactiei
  pri: int,                             # prioritatea tranzactiei
  shmId, semId: int,                    # identificatori pentru memoria partajata si pentru semafoare
  objName[1:MAX_OBJ_LEN]: char,         # numele obiectului (folosit pentru recuperare obiectelor dintr-o tr.)
  aplName[1:MAX_APL_NAME]: char         # numele aplicatiei
)
 type RECOVER = record of               # mesaje folosite de obiecte la recuperare
  ( type: int,                          # tipul mesajului
   transNo: int,                        # identificatorul tranzactiei
   objName[1:MAX_OBJ_LEN]: char,        # numele obiectului (folosit pentru recuperare obiectelor dintr-o tr.)
  )
typedef  CD_PACKET = record of          # mesaje folosite introducerea sau eliminarea unui nod din sistem
   (type: int                           # CONNECT, DISCONNECT sau GET_OBJECT
    host:  ADDR                         # adresa nodului asociat
    port: int                           # portul asociat la CS
   )
typedef  RD_OBJECT = record of # mesaje folosite introducerea sau eliminarea unui obiect din sist.
   (type: int                           # RECORD sau DELETE
    host:  ADDR                         # adresa nodului asociat
    port: int                           # portul asociat la CS obiectului
    pri: int                            # insa folosit si la implicarea unui nod
    aplName[1:MAX_APL_NAME]: char       # numele aplicatiei
    objName[1:MAX_OBJ_LEN]: char,       # numele obiectului
   )
type BT_OBJECT= record of               # structura unui pachet transmis la implicarea unui obiect
 (type: int,                            # tipul mesajului
  errCode: int,                         # codul de eroare
  transId: int,                         # identificatorul tranzactiei
  pri: int,                             # prioritatea tranzactiei
  shmId, semId: int,                    # identificatori pentru memoria partajata si pentru semafoare
  semNo: int                            # numarul semaforului din set rezervat pentru obiect
 aplName[1:MAX_APL_NAME]: char          # numele aplicatiei
 )
typedef  ACK_ID_TRANS = record of       # mesaje folosite pentru determinarea prioritatii si
   (type: int                           # identificatorului unei prioritati
    tranId: int                         # identificator
    pri: int,                           # numele obiectului
   )
# canale statice
chan chanf_APL( BT_ACK_PACKET)          # canal static pentru mesajele destinate aplicatiei
chan chanf_TM(BT_PACKET)                # canal static pentru mesajele destinate managerului de tranzactii
chan chanf_CS(kind;                     # canal static pentru mesaje la serverul de comunicatii:
  case TM_BEGIN_TRANSACTION: BT_ACK_PACKET      # TM_BEGIN_TRANSACTION de la TM
  case  RECORD, DELETE:RD_PACKET) # RECORD,  DELETE  de la managerii de obiecte locale
chan chans_CS()                         # canal static pentru mesajele destinate serverului de comunicatii
chan chanu_CS()                         # canal static pentru mesajele destinate serverului de comunicatii
chan chans_CM()                         # canal static pentru mesajele destinate managerului central
```


BUPT

```
chan chanu_NS()                          # canal static pentru mesajele destinate serverului de nume
chan chanu_OBJ(BT_OBJECT)                # canal static cu mesaje la initierea si terminarea tranzactiior
# canale dinamice pentru comunicatie prin memorie partajata
chan chan_S_APL (PACKET)                 #canal dinamic pentru mesajele destinate aplicatiei
chan chan_S_TM ( PACKET)                 # canal dinamic pentru mesajele destinate man. de tranzactii fiu
chan chan_S_CS (PACKET)                  #canal dinamic pentru mesajele destinate serv. de comunicatii fiu
# canale dinamice tip socluri unix
chan chan_U_TM ( RECOVER)                # canal dinamic pentru mesajele destinate man. de tranzactii fiu
chan chan_U_CS (BT_ACK_PACKET)           #canal dinamic pentru mesajele destinate serv. de comunicatii fiu
chan chan_U_NS (CD_PACKET)
#canale dinamice tip soclu inet
chan chan_K_CS(kind;           # canal dinamic (soclu inet) pentru mesaje la serverul de comunicatii:
  case NETWORK_BEGIN_TRANSACTION: BT_ACK_PACKET       # NETWORK de la TM
  case RECORD, DELETE: RD_PACKET) # RECORD, DELETE de la managerii de obiecte locale
chan chan_K_CM(kind;           # canal static (soclu inet)pentru mesaje la serverul de comunicatii:
  case NETWORK_BEGIN_TRANSACTION: BT_ACK_PACKET       # NETWORK de la TM
  case RECORD, DELETE: RD_PACKET) # RECORD, DELETE de la managerii de obiecte locale
chan chan_KO_CS (BT_OBJECT), chan_KT_CS (BT_OBJECT)       # pt comunicatie CS-CS


var slotsTab[1: MAX_SLOTS]: PACKET     # tabela de sloturi pentru pachetele comunicate prin mem. partajata
var S_SHM, S_APL, S_TM, S_CS, S_Child, S_Free_Slot: semaphore # semafoare pentru acces la mem. part.
```

9.4.2. Integrarea proceselor de aplicaţie în sistemul DOSTP

Setul de funcţii prezentat la 9.1 realizează integrarea proceselor de aplicaţie în sistemul DOSTP, integrare transparentă pentru utilizatori. Aceste funcţii sunt incluse într-o bibliotecă cu care se linkeditează aplicaţiile. Ele pot fi descrise astfel:

```
const MAX_TAB = MAX_SLOTS - 2          # numarul maxim de cereri asincrone care pot fi emise de aplicatie
const SYNC_RES_SLOT = MAX_TAB          # slot rezervat pentru cererile sincrone
type REQUEST_PACKET = record of        # cerere de efectuare a unei operatii asupra unui obiect
        (empty : bool                  # pachetul este sau nu este folosit
         seqNo : int                   # numarul cererii în cadrul tranzactiei
         perrorCode : pointer to int   # memoreaza adresa la care va fi returnat codul de eroare
         resAdr : pointer to char      # memoreza adresa la care va fi returnat raspunsul la cerere
         slotUsage : int               # contor pentru folosirea pachetului
         timeOut : time                # timpul dupa care nu se mai asteapta raspunsul
        )
var reqTab[1:MAX_TAB]: REQUEST_PACKET          # tabela de cereri asincrone
var validDOSTP: bool = FALSE                   # indica daca aplicatia s-a conectat la DOSTP
var validTrans: bool = FALSE                   # indica daca o tranzactie este activa
var alreadyTrans: bool = FALSE                 # indica daca aplicatia a initiat o tranzactie
var oldTrSucc: bool =true          # o noua prioritate (de la managerul central) pentru o tranzactie noua
var oldPri = 0                     # prioritatea ultimei tranzactii executate
var asyncReq = 0: int              # numarul cererii in cadrul tranzactiei
var seqNo: int                     # numerele de secventa ale pachetelor
var pckt: PACKET                   # pentru trimiterea pachetelor in memoria partajata
var errCode : int                  # retine codurile de eroare returnate de functii
var errReqCode: int                # retine codul de eroare pentru operatii
var slot: int                      # pentru a retine numarului locatiei pachetului in memoria partajata

InitDOSTP[1:MAX_APL_LEN]: char) ::
        if name = sir nul → return EBADF
        if validDOSTP → return EDOSTP_INITIALIZE_DONE
        genereaza numele aplicatiei in aplName
        creeaza canalul static de comunicatie cu numele aplName.FIFO si obtine identitatea lui in chanf_APL
        if chanf_APL nu a putut fi creat → return ECREATE_APL_FIFO
        validDOSTP := true;
```


BUPT

```
        return SUCCES

BeginTransaction()::
    var btPckt: BT_PACKET              # pachet trimis la initierea unei tranzactii
    var btAckPckt: BT_ACK_PACKET       # pachet trimis pentru confirmarea initierii unei tranzactii
        if validDOSTP = FALSE return EDOSTP_INITIALIZE_EXPECTED
        if alreadyTrans → return EBEGIN_TRANS_DONE
        fa i = 0 to MAX_TAB +1 → reqTab [i].empty := 1; reqTab[i].slotUsage := 0; af      # initializari
        genereaza numele managerului de tranzactii in TMName.FIFO
        deschide canalul chanf_TM (cu numele TMName.FIFO) pentru comunicatia aplicatie -> TM
        if chanf_TM nu a putut fi deschis → return EOPEN_TM_FIFO
        deschide canalul chanf_APL pentru comunicatia TM ->aplicatie
        if chanf_APL nu a putut fi deschis → return EOPEN_APL_FIFO
        btPckt.type := APL_BEGIN_TRANSACTION ;
        if  oldTrSucc = FALSE  → btPckt.pr := -1    # prioritate noua pentru tranzactie
        []  oldTrSucc = true → btPckt.pr := oldPri # daca se reia o tranzactie se retine vechea prioritate
        fi
        if ((errCode := send chanf_TM(btPckt)) ≠ SUCCESS) → return EWRITE_TM_FIFO_FAIL
        programeaza un semnal de alarma TIME_OUT_APL_TM pentru receptia pe canalul chanf_APL
        if ((errCode := receive chanf_APL(btAckPckt) ≠ SUCCESS) return EREAD_APL_FIFO_FAIL
        inchide canalele chanf_APL, chanf_TM
        if btAckPckt.errCode ≠ SUCCESS → return btAckPckt.errCode
        [] btAckPckt.errCode = SUCCESS →
                oldPri := btAckPckt.pri; semId := btAckPckt.semId ; shmId :=btAckPckt.shmId
                if nu se poate realiza atasarea la memoria partajata → return EACCESS_SHM_FAIL
                [] se poate realiza atasarea la memoria partajata →
                        validTrans := true ; alreadyTrans := true; asyncReq := 0;     # initializari
                        return SUCCESS
                fi
        fi

SyncRequest (objectName [1: MAX_OBJ_NAME], sourceAddr: pointer to char,
                        destAddr: pointer to char, timeOut: int)::
        if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
        if validTrans = FALSE → return EINVALID_TRANS
        if objectName = sir nul → return EBADF
        if asyncReq > MAX_TAB → return ENO_BUF
        if objectName ≠ sir nul → pckt.type := REQUEST; pckt.objName := objectName fi
        pckt.seqNo := seqNo; pckt.aplSlot := SYNC_RES_SLOT; pckt. text := sourceAddr
        pckt.aplSlot Usage := reqTab [SYNC_RES_SLOT].SlotUsage;
        se completeaza o cerere reqTab [SYNC_RES_SLOT] cu valorile:.resAdr:= destAddr, .seqNo:= seqNo,
                                        .perrorCode:= errReqCode, .empty:=FALSE, .timeOut:= timeOut
        send chan_S_TM(slot)
        programeaza un semnal de alarma pentru receptia pe chan_S_APL pentru timeOut sec
        again:  errCode := receive chan_S_APL(slot);
                if errCode = SUCCES →
                  if slotsTab[slot]. aplSlot = SYNC_RES_SLOT →
                        if reqTab [SYNC_RES_SLOT].slotUsage = slotsTab[slot]. aplSlotUsage →
                                i = slotsTab[slot]. aplSlot
                                completeaza la reqTab [i].resAdr si reqTab [i].perrorCode
                                cu slotsTab[slot].text, slotsTab[slot].errCode  # s-a rezolvat raspunsul
                                elibereaza slot                                 # pentru cererea sincrona
                                reqTab [i]slotUsage:= reqTab [i].slotUsage+1
                        [] reqTab [SYNC_RES_SLOT].slotUsage ≠ slotsTab[slot]. aplSlotUsage →
                                elibereaza slot          # pachet sincron intirziat; se ignora
                                goto again
```

```
                fi
          [] slotsTab[slot]. aplSlot ≠ SYNC_RES_SLOT →
             i = slotsTab[slot]. aplSlot
             if reqTab[i].empty = FALSE and reqTab[i].slotUsage = slotsTab[slot]. aplSlotUsage→
                         completeaza la reqTab [i].resAdr si reqTab [i].perrorCode
                         cu slotsTab[slot].text, slotsTab[slot].errCode # s-a rezolvat raspunsul
                         asyncReq := asyncReq - 1                       # pentru cererea asincrona
                         reqTab [i]slotUsage:= reqTab [i].slotUsage+1
             fi
             elibereaza slot
             goto again
          fi
        [] errCode = TIME_OUT →
                reqTab [SYNC_RES_SLOT].SlotUsage=reqTab [SYNC_RES_SLOT].SlotUsage + 1
                seqNo := seqNo + 1; reqTab [SYNC_RES_SLOT].empty:=true
                reqTab [SYNC_RES_SLOT].perrorCode := TIME_OUT
                return TIME_OUT
        fi
        seqNo := seqNo +1
        if reqTab [SYNC_RES_SLOT].perrorCode SUCCESS → return COMMIT_ERROR
        [] return SUCCESS

AsyncRequest (objectName [1: MAX_OBJ_NAME]: char, sourceAddr: pointer to char,
                       destAddr: pointer to char, perrCode: pointer to int, timeOut: int)::
    if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
    if validTrans = FALSE → return EINVALID_TRANS
    if objectName = sir nul → return EBADF
    if asyncReq > MAX_TAB → return ENO_BUF
    if nu exista o intrare libera i in reqTab → return ENO_BUF
    if objectName ≠ sir nul → pckt.type := REQUEST; pckt.objName := objectName fi
    pckt.seqNo := seqNo; pckt.aplSlot := i; pckt. text := sourceAddr ;
    pckt.aplSlot Usage := reqTab [i].SlotUsage;
    se completeaza o cerere reqTab [i] cu valorile:
    .resAdr:= destAddr, .seqNo:= seqNo, .perrorCode:= errReqCode, .empty:=1, .timeOut:= timeOut
    send chan_S_TM(slot)
    seqNo := seqNo +1; asyncReq := asyncReq + 1
    return SUCCESS

Synchronize( )::
    if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
    if validTrans = FALSE → return EINVALID_TRANS
    if asyncReq = 0 → return SUCCESS
    determina in maxTimeOut maximum dintre toate valorile reqTab[i].timeOut
    if maxTimeOut > timpul curent →
            programeaza un semnal de alarma peste maxTimeOut - timpul curent
            again: errCode := receive chan_S_APL(slot);
            if errCode ≠ TIME_OUT →
               i = slotsTab[slot]. aplSlot
               if reqTab[i].empty = FALSE and reqTab[i].slotUsage = slotsTab[slot]. aplSlotUsage→
                         completeaza la reqTab [i].resAdr si reqTab [i].perrorCode
                         cu slotsTab[slot].text, slotsTab[slot].errCode # s-a rezolvat raspunsul
                         asyncReq := asyncReq - 1                       # pentru cererea asincrona
                         reqTab [i]slotUsage:= reqTab [i].slotUsage+1
               fi
               elibereaza slot
               goto again
```


BUPT

```
            fi
    fi
    fa i:=1 to MAX_TAB st reqTab[i].empty = FALSE
            reqTab[i].empty = true ;reqTab [i]slotUsage:= reqTab [i].slotUsage+1
            reqTab [i].perrorCode := TIME_OUT; asyncReq := asyncReq - 1
    af
    return SUCCESS
```

EndTransaction( )::

```
    if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
    if validTrans = FALSE → return EINVALID_TRANS
    if asyncReq >0 → return ESINCRONIZE_EXPECTED
    pckt.type := END_TRANSACTION
    errCode = SyncRequest_S(NULL, pckt, pckt, TIME_OUT_APL_TM_SREQ );
    if errCode = ETIME_OUT oldTrSucc := FALSE
    [] errCode ≠ ETIME_OUT oldTrSucc := true
    fi
    if errCode ≠ SUCCESS →
            if errCode = ETIME_OUT → return ETIME_OUT
            [] errCode ≠ ETIME_OUT → errCode = ECOMMIT_FAIL
    pckt.type := APL_END
    errCode = AsyncRequest_S(NULL, pckt, pckt, errCode, TIME_OUT_APL_TM_AREQ );
    alreadyTrans := FALSE
    if errCode ≠ SUCCESS →
            if errCode = ETIME_OUT → return ETIME_OUT
            [] errCode ≠ ETIME_OUT → errCode = ECOMMIT_FAIL
    return SUCCESS
```

AbortTransaction( )::

```
    if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
    if validTrans = FALSE → return EINVALID_TRANS
    pckt.type := ABORT_TRANSACTION
    errCode = AsyncRequest_S(NULL, pckt, pckt, errCode, TIME_OUT_APL_TM_AREQ );
    alreadyTrans := FALSE; oldTrSucc := FALSE;
    inchide canalele chanf_APL, chanf_TM
    return SUCCESS
```

CloseDOSTP( )::

```
    if validDOSTP = FALSE → return EDOSTP_INITIALIZE_EXPECTED
    inchide canalele chanf_APL, chanf_TM; sterge chanf_APL
    validDOSTP = FALSE
    return SUCCESS
```

9.4.3. Proiectarea managerilor de tranzactii

Aplicațiile trimit cererile de operații obiectelor prin intermediul managerilor de tranzacții. Un manager de tranzacție crează cîte un proces fiu, TM-T*i* pentru fiecare tranzacție *i* pe care o inițiază o aplicație. Un manager de tranzacție se constituie deci în mai multe procese: procesul principal TM și procesele fii, care supervizează tranzacțiile. Procesul TM asteaptă într-o buclă continuă receptionarea mesajelor de inițiere ale tranzacțiilor de la aplicații, și recepționarea mesajelor de acceptare a unei tranzacții sau terminare a unei tranzacții de la procesele fii TM-T*i*. Procesul TM ține evidența tranzacțiilor pastrind liste interne cu

BUPT

corespondenţa identificator tranzacţie - nume aplicaţie; listele sunt actualizate la începerea sau terminarea unei tranzacţii. Identificatorul de tranzacţie, ca şi prioritatea tranzacţiilor,se obţin prin contactarea managerului central. Pe baza priorităţii unei tranzacţii, managerii de obiecte decid, în cazul unui conflict de acces, care dintre tranzacţiile în conflict se va termina cu succes.

Procesul TM recepţionează mesaje într-un singur canal, *chan-TM*, creat static, la lansarea sa în execuţie. Cererile de operaţii asupra obiectelor sunt recepţionate de procesele TM-T*i* în zona de memorie partajată creată de TM-T*i* la lansarea în execuţie; mecanismul de recepţie a unui mesaj din zona de memorie partajată va fi reprezentat mai jos, în descrierea managerilor de tranzacţie, prin recepţionarea unui mesaj din canalul guvernat de semaforul asociat procesului, *chan-S-TM* [*Ti*]. Procesele TM-T*i* constau într-o buclă principală în care se receptionează mesaje din zona de memorie partajată (cereri de operaţii de la aplicaţie sau răspunsuri de la cereri anterioare, răspunsuri trimise de serverul de comunicaţie sau de managerii de obiecte locale).

Trebuie precizat că, la lansarea în execuţie, într-o fază de iniţializare, înainte de a intra.în bucla de receptionare a mesajelor, managerul de tranzacţii (TM) analizează fişierele sale de log pentru a detecta dacă execuţia sa are loc după o terminare anormală, survenită în urma defectării nodului. La nivelul managerului de tranzacţii, TM, există două fişiere log: primul fişier (TMREC1) memorează identificatorii şi starea tranzacţiilor, iar cel de-al doilea, (TMREC2) memorează pentru fiecare tranzacţie numele obiectelor implicate în tranzacţie şi starea obiectului în tranzacţie. O tranzacţie se poate gasi în fişierul TMREC1 în una din urmatoarele patru stări: (1) tranzacţia nu a ajuns la *EndTransaction()* sau *AbortTransaction()* (WORKING); (2) s-a executat faza *prepare to commit*, dar nu şi *commit* pentru tranzacţie (PREPARE-TO-COMMIT) şi deci nu se cunosc răspunsurile managerilor de obiecte implicaţi în tranzacţie; (3) s-a executat faza COMMIT, în urma primirii răspunsurilor afirmative de la toţi managerii de obiecte (COMMIT); (4) s-a executat faza ABORT (ABORT). Stările poibile ale obiectelor din tranzacţii, înregistrate în fişierul TMREC2 sunt: (1) obiectul recepţionează cereri şi le tratează (BUSY); (2) obiectul a răspuns afirmativ la un mesaj PREPARE-TO-COMMIT (OK); (3) obiectul a răspuns negativ la un mesaj PREPARE-TO-COMMIT, constatînd un conflict cu o altă tranzacţie mai prioritară, FAIL.

După ce managerii de obiecte au răspuns cu ACK-ABORT sau ACK-COMMIT intrările corespunzatoare tranzacţiei respective din TMREC1 şi TMREC2 se invalidează (se modifică numarul tranzacţiei la zero, numar invalid, lasînd celelalte informatii neschimbate).

În faza de initializare, în urma analizei fişierului TMREC1, managerul de tranzacţii TM crează cîte un proces pentru fiecare tranzacţie a cărei intrare este neştearsă; procesul fiu, pe baza informaţiilor din TMREC2 şi în funcţie de starea tranzacţiei va asigura recuperarea tranzacţiei astfel:

(1) Pentru o tranzacţie gasită în starea WORKING se trimite la toţi managerii de obiecte (găsiţi în TMREC2) comanda ABORT.

(2) Pentru o tranzacţie gasită în starea PREPARE trebuie verificate răspunsurile managerilor de obiecte implicaţi în tranzacţie (din fişierul TMREC2); dacă toţi managerii au răspuns OK, atunci se trimite comanda COMMIT; daca cel puţin un manager a răspuns cu FAIL atunci se trimite ABORT.

(3) Pentru o tranzacţie gasită în starea COMMIT se transmite COMMMIT.

(4) Pentru o tranzacţie gasită în starea ABORT se transmite ABORT.

Dacă, pe de altă parte, în timpul desfăşurării tranzacţiei apare o defecţiune a unui nod care conţine un obiect implicat în tranzacţie, atunci serverul de comunicaţii detectează acest lucru şi sesizează managerul de comunicaţii. Acesta va consulta fişierele de recuperare şi va decide acţiunea de executat astfel: pentru tranzacţiile în starea COMMIT se transmite COMMIT, pentru tranzacţiile în orice stare diferită de COMMIT se transmite ABORT. Managerii de tranzacţii pot fi descrişi astfel:

```
enum TRANS_STATE = (ABORT, WORKING, PREPARE_TO_COMMIT, COMMIT)  # starile unei tranzactii
enum OBJ_RESPONSE = (BUSY, OK , FAIL)           # raspunsurile unui obiect la prepare-to-commit
type TMREC1_ART = record of                     # structura unui articol din fisierul TMREC1
        (transNo: int,
         transState: TRANS_STATE,
         transName[1: MAX_APL_LEN]: char
        )
type TMREC2_ART = record of                     # structura unui articol din fisierul TMREC2
        (transNo: int,
         objResp: OBJECT_RESPONSE,
         objectName[1: MAX_OBJ_LEN]: char
        )
type OBJECTS =  record of
        (free: bool,                            # obiect nefolosit cind free este true
         errCode: int,
         objTransId :int,
         ackPrToCom, ackCom, ackAbort: int
         objectName[1: MAX_OBJ_LEN]: char
```


BUPT

```
        )
TransactionManager::
   var TMREC1: file of TMREC1_ART              # fisierele de recuperare
   var TMREC2: file of TMREC2_ART
   var mes: BT_PACKET                          #  mesaj primit pe canalul chanf_TM
   var S_Rec1, S_Rec2: semaphore := (1,1)      # semafoare pentru acces eaxclusiv la fisierele log
   var semId: int                              # identificatorul setului de seamfoare
   var mes: BT_ACK_PACKET      # pentru transmisia mesajelor pe canale statice
   var semIdR : semaphore      # folosit pentru contorizarea fiilor care au efectuat recuperarea obiectelor
  var shmId: int                      # identificator al zonei de memorie partajata
  var objTab: OBJECTS[1:MAX_OBJ_NO] # tabela cu informatii despre obiectele implicate in tranzactie
  var objNo: int                      # numarul de obiecte din tranzactie
  var previous_abort: int

        # faza de initializare
        programeaza tratarea semnalului SIGCHLD
        if nu se pot deschide semafoarele S_Rec1, S_Rec2 identificate prin semId pentru controlul accesului
                                    la fisierele de recuperare  → Panic()
        fi
        if nu se poate crea si deschide canalul static pentru receptia mesajelor chanf_TM  → Panic() fi
        if nu se pot deschide fisierele de recuperare  → Panic() fi
        # faza recuperare dupa repunerea in functiune a managerului de tranzactii
        do not eof (TMREC1)
            if exista in fisierul TMREC1 o tranzactie neinvalidata →
              lanseaza in executie un proces fiu (fork+execlp) TransactionManagerServer caruia i se transmite:
                    transName, semId, transNo, transState,0
             fi
        od
        # faza de tratare a cererilor de la aplicatii
        do true  →
             receive chanf_TM (mes)
            if mes.type = APL_BEGIN_TRANSACTION →
               lanseaza in executie un fiu (fork+execlp) TransactionManagerServer caruia i se transmite:
                     mes.aplName, semId, 0, 0, mes.pri,
            [] mes.type = TRANSACTION_ACCEPTED  →
                  insereaza tranzactia in lista de tranzactii active; pentru o tranzactie se memoreaza:
                     mes.aplName,  mes.pri, mes. transId,
            [] mes.type = TRANSACTION_ENDED  → elimina tranzactia din lista de tranzactii active
            fi
        od

TransactionMangerServer(aplName[MAX_APL_LEN]:char, semId: int, transNo: int, transState, pri: int)::
var waiting: bool               # folosita pentru a indica validitatea tranzactiei
var childNo: int                # contorizeaza numarul de procese fii create pentru recuperare
var recover : RECOVER           # pentru mesajele primite de la obiecte la recuperare
var rec: int, errCode: int      # variabila de lucru
var pckt, pcktr: PACKET
var transId: int; priority: int, objId: int, status: int       # variabile de lucru
var seqNoEnd: int, aplSlotUsage: int, aplSlot: int            # variabile de lucru
var ackPrNo: int, csFlag: int, aplFlag: int, resp: int         # variabile de lucru

    if  nu se poate determina din fisierul de configurare adresa managerului central  → Panic() fi
    programeaza tratarea semnalelor SIGCHLD si SIGALRM
    if  nu se poate deschide canalul de comunicatie cu CS chanf_CS  → Panic() fi
    if transNo ≠ 0  →      # faza de recuperare a tranzactiilor ramase neterminate de la ultima executie
        if  nu se poate deschide  semaforul semIdR  initializat la 0  → Panic() fi
        if  transState = ABORT or transState = WORKING →
            modifica starea tranzactiei din fisierul TMREC1 la ABORT
            if nu se poate deschide fisierul TMREC2 → Panic() fi
```


BUPT

```
            childNo := 0
            do cit timp exista un obiect objName implicat in tranzactia transNo →
               childNo := childNo +1
               TransactionRecovery(ABORT, transNo, pri, objName, semIdR)
            od
            do cit timp valoarea semaforului semIdR ≠ childNo → od   # se asteapta terminarea tuturor fiilor
            invalideaza in TMREC1 tranzactia transId
        [] transState = PREPARE_TO_COMMIT →
            if nu se poate deschide fisierul TMREC2 → Panic() fi
            if toate obiectele tranzactiei transId din TMREC2 au raspuns OK →
                  rec:= COMMIT; modifica starea tranzactiei in TMREC1 la COMMIT
            [] nu toate obiectele tranzactiei transId din TMREC2 au raspuns OK →
                  rec := ABORT; modifica starea tranzactiei in TMREC1 la ABORT
            childNo := 0
            do cit timp exista un obiect objName implicat in tranzactia transNo →
                  childNo := childNo +1
                  TransactionRecovery(rec, transNo, pri, objName, semIdR)
            od
            do cit timp valoarea semaforului semIdR ≠ childNo → od   # se asteapta terminarea tuturor fiilor
            invalideaza in TMREC1 tranzactia transId
        [] transState = COMMIT →
            childNo := 0
            do cit timp exista un obiect objName implicat in tranzactia transNo →
               childNo := childNo +1
               TransactionRecovery(COMMIT, transNo, pri, objName, semIdR)
            od
            do cit timp valoarea semaforului semIdR ≠ childNo → od   # se asteapta terminarea tuturor fiilor
            invalideaza in TMREC1 tranzactia transId
        fi
        exit
    fi
    # faza de initializare
    if nu se poate deschide canalul static pentru comunicatia cu aplicatia chanf_APL → Panic() fi
    if nu se poate crea o zona de mem. partajata cu id. shmId → FatalError(ETM_CREATE_SHM) fi
    if eroare la atasarea la zona de mem. partajata la adr. pShm → FatalError(ETM_ACCESS_SHM_FAIL) fi
    if nu se poate crea setul de sem. semId pt. acc. la mem. partajata → FatalError(ETM_CREATE_SEM) fi
    se initializeaza tabela de sloturi
    if nu se poate deschide canalul de comunicatie chanf_TM → FatalError(ETM_OPEN_TM_FIFO) fi
    mes.type := TM_BEGIN_TRANSACTION; mes.errCode := SUCCESS; mes.aplName := aplName
    mes.shmId := shmId; mes.semId := semId; send chanf_CS(mes)
    programeaza un semnal de alarma pentru receptia pe chan_S_MT la TIME_OUT_TM_CS_BTACK
    errCode := receive chan_S_MT( slot)
    if errCode = TIME_OUT → FatalError(ECS_TIME_OUT) fi
    if slotsTab[slot].errCode ≠ SUCCESS → FatalError(slotsTab[slot].errCode) fi
    mes.type := ACK_APL_BEGIN_TRANSACTION; send chanf_APL(mes)     # raspuns la aplicatie
    transId := slotsTab[slot].transId; priority:= slotsTab[slot].priority
    mes.type := TRANSACTION_ACCEPTED; mes.transId:= transId; mes.pri:= priority
    send chanf_TM(mes)
    scrie in fisierul TMREC1 un articol pentru tranzactia initiata, avind cimpurile:
          .transNo:= transId; transState:= WORKING; .transName:= aplName;
    fa i= 1 to MAX_OBJ_NO    objTab.free := true            # initializeaza tabela de obiecte
    waiting = true
    # faza de receptionare a cererilor de operatii si a raspunsurior
    do  waiting = true →
    receive chan_S_TM(slot); pcktr:= slotTab[slot]
    if pcktr.type = REQUEST→
          if pcktr.objName exista in pozitia i in objTab[i]   objId := i →
          [] objId := -1
          fi
```


BUPT

```
            if objId ≠ -1 →                      # nu este prima cerere pentru acest obiect in tranzactie
                  slotTab[slot].transId:= transId; slotTab[slot].objId:= objId; send chan_S_CS (slot)
            [] objId = -1 →                      # este prima cerere pentru acest obiect in tranzactie
                  if nu mai exista loc in tabela de obiecte objTab →
                        slotTab[slot].errCode:= ETOO_MANY_OBJECTS; send chan_S_APL(slot)
                  [] insereaza obiectul in objTab intr-o pozitie libera objId
                        objNo := objNo +1; slotTab[slot].:= FIRST_REQUEST
                        slotTab[slot]..transId:= transId; slotTab[slot].objId:= objId; send chan_S_CS (slot)
                        Down(S_Rec2)
                        scrie in fisierul TMREC2 un articol pentru noul obiect cu cimpurile:
                              .objName:= slotTab[slot].objName, .transId:= transId, resp:= BUSY
                        Up(S_Rec2)
                  fi
            fi
      [] pcktr.type = ACK_FIRST_REQUEST→
            send chan_S_APL(slot);
            if slotTab[slot].errCode = EOBJ_NOT_FOUND →
                  sterge din objTab intrarea obiectului slotTab[slot].objName;  objNo := objNo - 1
                  Down(S_Rec2)
                  sterge  din fisierul TMREC2 articolul  obiectului slotTab[slot].objName
                  Up(S_Rec2)
                  aloca un nou slot slot1 in slotsTab; slotTab[slot1].type:= ABORT
                  AbortTransaction(slot1);                  # forteaza terminarea anormala a tranzactiei
      [] pcktr.type = ACK_REQUEST→
            send chan_S_APL(slot);
      [] pcktr.type = ABORT_TRANSACTION→
            AbortTransaction(slot);
      [] pcktr.type = ACK_ABORT_TRANSACTION→
            objTab[pcktr.objId].ackAbort :=  1
                  sterge din objTab intrarea obiectului slotTab[slot].objName;  objNo := objNo - 1
                  Down(S_Rec2);sterge din TMREC2 articolul obiectului slotTab[slot].objName;  Up(S_Rec2)
                  elibereaza slot
                  if objNo = 0 →
                        aloca slot1; slotTab[slot1].type := END_TRANSACTION; send chan_S_CS (slot1)
                        again:errCode := receive chan_S_TM(slot)
                        if errCode = SUCCESS  and slotsTab[slot].type ≠ CS_END  →
                              elibereaza slot; goto again:
                        [] errCode = TIME_OUT→
                              aloca slot; slotTab[slot].type := ABORT_TRANSACTION
                              AbortTransaction(slot)
                        fi
                        End()
                  [] Up(S_Child)
                  fi
      [] pcktr.type = END_TRANSACTION→
            seqNoEnd:= pcktr.seqNo;  aplSlotUsage: = pcktr.aplSlotUsage;  aplSlot: = pcktr.aplSlot;
            if  objNo ≠ 0 →
                  if freePackListIdx ≠ 1 → End() fi   #in mod normal trebuie sa existe numai pachetul End
                  elibereaza slot;
                  # faza prepare_to_commit
                  initializeaza  pentru toate obiectele objTab cimpurile .ackPrToCom :=0, .errCode := 0
                  initializeaza semaforul S_Child la MAX_SLOTS - 1; ackPrNo := 0;
                  se creaza un proces fiu care transmite mesajul la toate obiectele; el executa:
                        Down(S_Rec1);
                        modifica in TMREC1 starea tranz. .transState:=ABORT;
                        Up(S_Rec1)
                        fa obiectele i st objTab[i].free = false →
                          Down(S_Child);  aloca slot1; slotTab[slot1].type := PREPARE_TO_COMMIT
```


BUPT

```
                slotTab[slot1].objId:= i;slotTab[slot1].objTransId:= transId;slotTab[slot1].errCode=0
          slotTab[slot1].objName:= objectTab[i].; send chan_S_CS (slot1)
                af
                exit
    [] objNo = 0 →
          status: = 0
          fa obiectele i st objTab[i].free = false →
                if objTab[i].errCode ≠ SUCESS → status := objTab[i].errCode fi
          af
          slotTab[slot].errCode := status; slotTab[slot]:seqNo= seqNoEnd;
          slotTab[slot].aplSlotUsage:= aplSlotUsage; slotTab[slot].aplSlot:= aplSlot;
          send chan_S_APL(slot)
          aloca slot1; slotTab[slot1].type := END_TRANSACTION; send chan_S_CS (slot1)
          csFlag := 0;
          do csFlag=0 →
                receive chan_S_TM(slot)
                if slotTab[slot].type = CS_END → csFlag=1
                [] slotTab[slot].type = APL_END → aplFlag=1
                elibereaza slot
          od
          do aplFlag=0 →
                receive chan_S_TM(slot)
                if slotTab[slot].type = APL_END → aplFlag=1
                elibereaza slot
          od
          End()
[] pcktr.type = ACK_PREPARE_TO_COMMIT →
    objTab[pcktr.objId].ackPrToCom := 1; resp:= OK;
    if pcktr.errCode ≠ SUCCESS → objTab[pcktr.objId].errCode := pcktr.errCode
    [] objTab[pcktr.objId].errCode := text.errCode
    fi
    if objTab[pcktr.objId].errCode ≠ SUCCESS → resp :=FAIL fi
    ackPrNo := ackPrNo +1
    Down(S_Rec2);modifica in fis TMREC2 articolul pentru obiect la cimpul .resp:= resp; Up(S_Rec2)
    if resp ≠ OK → modifica in fisierul TMREC1 articolul pentru obiect in cimpul .tranState:= ABORT fi
    elibereaza slot
    if ackPrNo = objNo →
          if toate obiectele au raspuns cu OK → status := OK
          [] nu toate obiectele au raspuns cu OK → status := FAIL
          if status ≠ OK →
                rezerva slot
                slotTab[slot].errCode := status; slotTab[slot]:seqNo= seqNoEnd;
                slotTab[slot].aplSlotUsage:= aplSlotUsage; slotTab[slot].aplSlot:= aplSlot;
                send chan_S_APL(slot)
                aloca slot1; slotTab[slot1].type := ABORT_TRANSACTION;
                AbortTransaction(slot1)
          [] if status = OK → # faza de commit
              initializeaza pentru toate obiectele objTab cimpurile .ackPrToCom :=0, .errCode := 0
              initializeaza semaforul S_Child la MAX_SLOTS - 1
              se creaza un proces fiu care transmite mesajul la toate obiectele; el executa:
                 Down(S_Rec1);modifica in TMREC1 starea tranz. .transState:=ABORT; Up(S_Rec1)
                 fa obiectele i st objTab[i].free = false →
                   Down(S_Child); aloca slot1; slotTab[slot1].type := COMMIT
                   slotTab[slot1].objId:= i;slotTab[slot1].objTransId:= transId;slotTab[slot1].errCode=0
          slotTab[slot1].objName:= objectTab[i].; send chan_S_CS (slot1)
                 af
                 exit
          fi
```


BUPT

```
            [] Up(S_Child)
        [] pcktr.type =  ACK_COMMIT →
            objNo := objNo - 1; elibereaza slot
            if objNo = 0 →
                    if toate obiectele au raspuns cu OK → status := OK
                    [] nu toate obiectele au raspuns cu OK → status := FAIL
                    fa toate obiectele din objTab st objTab.free = false →
                            elibereaza intrarea
                            Down(S_Rec2);sterge din TMREC2 articolul obiectului Up(S_Rec2)
                    af
                    rezerva slot; slotTab[slot].errCode := status; slotTab[slot]:seqNo= seqNoEnd;
                    slotTab[slot].aplSlotUsage:= aplSlotUsage; slotTab[slot].aplSlot:= aplSlot;
                    send chan_S_APL(slot)
                    aloca slot1; slotTab[slot1].type := END_TRANSACTION;
                    send chan_S_CS(slot)
                    csFlag := 0;
                    do csFlag=0 →
                            receive chan_S_TM(slot)
                            if slotTab[slot].type = CS_END → csFlag=1
                            [] slotTab[slot].type = APL_END → aplFlag=1
                            elibereaza slot
                    od
                    do aplFlag=0 →
                            receive chan_S_TM(slot)
                            if  slotTab[slot].type = APL_END → aplFlag=1
                            elibereaza slot
                    od
                    End()
            [] Up(S_Child)
        [] pcktr.type =  APL_END → aplFlag := 1
        [] pcktr.type =  CS_END → aplFlag := 1 waiting := false
        [] pcktr.type =  INTERRUPT →
            status := starea tranzactiei citita din TMREC1
            if status ≠ COMMIT   →  pcktr.type := INTERRUPT_ABORT
            [] status = COMMIT   →  pcktr.type := INTERRUPT_COMMIT
            fi
            AbortTransaction(slot)
        fi
        od
AbortTransaction(slot: int)::
var type: int
            type := slotTab[slot].type; elibereaza slot
            if type = INTERRUPT_COMMIT → transState:= COMMIT
            []  transState:= ABORT;
            fi
            Down(S_Rec1);
             modifica in fisierul TMREC1 articolul pentru tranzactie la starea  transState
            Up(S_Rec1);
            programeaza un semnal de alarma peste TM_ABORT_TIME
            if previous_abort  = 0
                    previous_abort := previous_abort + 1
                    if  objNo = 0 →
                            aloca slot1; slotTab[slot1].type := END_TRANSACTION; send chan_S_CS (slot1)
                            again:errCode :=  receive chan_S_TM(slot)
                            if errCode = SUCCESS  and slotsTab[slot].type ≠ CS_END  →
                                    elibereaza slot; goto again:
                            [] errCode = TIME_OUT→
```


BUPT

```
                        aloca slot; slotTab[slot].type := ABORT_TRANSACTION
                        AbortTransaction(slot)
                    fi
                    End()
            [] objNo ≠ 0 →
                    initializeaza toate cimpurile .errrCode si .ackAbort din objTab la 0
                    initializeaza semaforul S_Child la MAX_SLOTS - 1
                    se creaza un proces fiu care transmite mesajul la toate obiectele; el executa:
                        fa obiectele i st objTab[i].free = false →
                            Down(S_Child);  aloca slot1; slotTab[slot1].type := type;
                            slotTab[slot1].objId:= i; slotTab[slot1].objTransId:= transId;
                            slotTab[slot1].objName:= objectTab[i].; send chan_S_CS (slot1)
                        af
                        exit
            fi
    fi
End()::
  invalideaza in fisierul TMREC1 articolul pentru tranzactia transId
  se inchid semafoarele semId, se elibereaza zona de memorie partajata pShm
  mes.type := TRANSACTION_ENDED; send chanf_TM(mes)
  exit
FatalError(errCode: int)::
  mes.type := ACK_APL_BEGIN_TRANSACTION; mes.errCode := errCode
  send chanf_APL(mes)
  se inchid semafoarele semId, se elibereaza zona de memorie partajata pShm
  exit
Recovery(mes.type: int, transNo: int, pri: int, objName: [1:MAX_OBJ_LEN]: char, semIdR: int)::
   receive  chan_U_TM (recover)
   if recover.type = ACK_ABORT or recover.type = ACK_COMMIT
        Down(S_Rec2);invalideaza in TMREC2 articolul corespunzator lui transNo, objName; Up(S_Rec2)
   fi
   Up (semIdR)
   inchide chan_U_CS, chan_U_TM
   exit()
TransactionRecovery(type: int, transNo: int, pri: int, objName: [1:MAX_OBJ_LEN]: char, semIdR: int)::
  genereaza mes de tip BT_ACK_PACKET cu cimpurile:
        .type := type, .transNo := transNo, objName := objName, aplName := aplName
  if nu se poate obtine o conexiune chan_U_CS,chan_U_TM de la canalul chanu_CS → Panic() fi
  send chan_U_CS (mes)
  lanseaza in executie un fiu care executa  Recovery(mes.type, transNo, pri, objName, semIdR)
  inchide chan_U_CS, chan_U_TM
  return
```

9.4.4. Proiectarea serverelor de comunicaţii

În fiecare nod conectat la sistemul DOSTP se execută cîte un server de comunicaţii. Acesta are două scopuri principale: (1) *asigură desfaşurarea tranzacţiilor.* La iniţierea unei tranzacţii, managerul de tranzacţii contactează serverul de comunicaţii; acesta creează un proces server fiu, care va gestiona exclusiv comunicaţiile tranzacţiei respective. Pentru fiecare obiect de la distanţă implicat în aceasta tranzacţie, procesul server fiu va crea un alt proces care va receptiona mesajele de la obiectul de la distanţă. Serverul de comunicaţie creat pentru a deservi o tranzacţie se foloseşte de serviciile serverului de nume pentru a determina localizarea obiectelor în sistem. Daca obiectul este local, transferul de mesaje se execută prin zona de memorie partajata. (2) *asigura notificarea sistemului în cazul pierderii legaturii cu un nod participant la o tranzactie.* Actiunile unui server de comunicaţii care detectează pierderea legăturii cu un nod participant la o tanzacţie depind de poziţia managerului de tranzacţii care a contactat serverul de comunicaţii. Daca acesta se află pe acelaşi nod, atunci serverul de comunicaţii anunţă managerul de tranzacţii că s-a pierdut legătura cu un nod implicat în tranzacţie şi asteaptă decizia lui (procesul fiu al serverului de comunicaţii care dialoga cu nodul respectiv este terminat); în caz contrar, serverul de comunicaţii anunta managerul de obiect că s-a pierdut legatura cu nodul coordonator şi se asteapta decizia coordonatorului, la repunerea sa în funcţiune. (vezi 10.2.)

Descrierea SR a serverului de comunicaţii este:

BUPT

```
const MAX_PROC =...                       # numarul maxim de obiecte de la distanta care pot fi implicate
const LOCAL = 1;const DISTANCE = 0        # indicatori pentru localizarea tranzactiilor si a obiectelor
typedef OBJECT = record of                # informatii despre obiectele implicate intr-o tranzactie
        (sem: int                  # numarul semaforului din setul semId asociat unei SHM
         local: int                       # obiect local sau nu
         chan: int                        # canalul (soclul) asociat pentru obiectele de la distanta
         name[1:MAX_OBJ_LEN]: char        # numele obiectului
        )

CommunicationServer()::
var semList[1:SEM_NO]: int        # numerele semafoarelor din setul semId asociate accesului la zona partajata
var semListIdx: int := FIRST_UNUSED      #  index in lista semList
var objTab[1:MAX_OBJ_NO]: OBJECT         # retine localizarea obiectelor dintr-o tranzactie
var semIdR: semaphore                    # semafor folosit la recuperare
var mes: CD_PACKET                       # pachet trimis la conectare CS  la  sistem
var typ:int                              # primeste tipul mesajelor
var pckRD: RD_PACKET              # pentru mesaje de la managerii de la obiecte
var pckBTAck: BT_ACK_PACKET              # pentru mesaje de la managerul central
var addrHost: ADDR                       # adresa nodului
var transErr: bool :=false               # indica faptul ca nu mai este valida  nici o cerere pt. tranzactie
var aplName[1:MAX_APL_NAME]: char        # variabile pentru memorarea informatiilor caracteristice tranz.
var shmId, semId: int                    # variabile de lucru care memoreaza identif. pentru SHM
var pidTab[1: MAX_PROC]: int             # identificatorii proceselor fii create

   # faza de initializare
   if nu se poate determina din fisierul de configurare canalul managerului central → Panic() fi
   programeaza tratarea semnalului SIGCHLD
   fa i = 1 to SEM_NO  semList[i] := i        # initializarea numerelor semafoarelor din set
   fa i = 1 to  MAX_OBJ_NO → objTab[i].local:= 0; objTab[i].chan:= -1  fi # initializarea tabelei de obiecte
   if  nu se poate crea si deschide canalul de comunicatie propriu chanf_CS  → Panic() fi
   if  nu se poate crea si deschide canalul de comunicatie propriu chans_CS  → Panic() fi
   if  nu se poate crea si deschide canalul de comunicatie propriu chanu_CS  → Panic() fi
   if nu se poate crea sem. semIdR pentru recuperare, cu valoarea initiala 0  → Panic() fi
   if nu obtine adresa nodului curent la addrHost  → Panic()  fi
   # faza de conectare la serverul central
   completeaza un mesaj mes cu cimpurile .type=CONNECT, .host= addrHost, .port = portul fixat pentru CS
   send chans_CM(mes)
   # receptie si tratare cereri locale pe chanf_CS
   if not empty (chanf_CS)  → receive chanf_CS(typ);
      if typ = RECORD or  typ = DELETE →
         receive chanf_CS(pckRD);
         creaza un proces fiu care executa:
            obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS; send chan_U_NS(pckRD);
            obtine o conexiune chan_K_CM, chan_K_CM de la chans_CM; send chan_K_CM(pckRD); exit
      [] typ =  TM_BEGIN_TRANSACTION →
         receive chanf_CS(pckBTAck);
               creaza un proces fiu care executa:  aplName := pckBTAck.aplName;
               instaleaza o rutina pentru SIGALRM; Services(LOCAL, pckBTAck); exit
      fi
   fi
   # receptie si tratare cereri de la distanta pe chans_CS
   if not empty (chans_CS)  →
      stabileste o conexiune chan_K, chan_K_CS;  receive chan_K_CS(typ);
      if typ = RECORD or  typ = DELETE →
         receive chan_K_CS(pckRD);
         creaza un proces fiu care executa:
               obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS; send chan_U_NS(pckRD); exit
      [] typ = RECORD_END →
         receive chan_K_CS(pckRD);
         creaza un proces fiu care executa:
```


BUPT

```
                obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS; send chan_U_NS(pckRD);
                receive chan_U_NS(pckRd); Up (semIdR); exit
        [] typ = NETWORK_BEGIN_TRANSACTION →
          receive chan_K_CS(pckBTAck);
                creaza un proces fiu care executa: aplName := pckBTAck.aplName;
                instaleaza o rutina pentru SIGALRM; Services(DISTANCE, pckBTAck); exit
        [] typ = ABORT or typ = COMMIT →
          receive chan_K_CS(pckBTAck);
                creaza un proces fiu care executa:
                        Down (semIdR); # asteapta sa se termine inregistrarea obiectelor;
                        Up (semIdR); Recover( pckBTAck, chan_K); exit
        fi
    fi
    # receptie si tratare cereri locale pe chanu_CS
    if not empty (chanu_CS) →
       stabileste o conexiune chan_U_CS, chan_U_TM; receive chan_U_CS(typ);
        if typ = ABORT or typ = COMMIT →
          receive chan_U_CS(pckBTAck);
                creaza un proces fiu care executa:
                        Down (semIdR); # asteapta sa se termine inregistrarea obiectelor;
                        Up (semIdR); Recover( pckBTAck,chan_U_TM); exit
        fi
    fi
Recover(data: BT_ACK_PACKET, chan_R:chan)::
var mes: RD_ OBJECT; dt: BT_OBJECT
        copiaza cimpurile comune de la data la dt
        completeaza mes cu valorile .type:= GET_OBJECT, .objName:= data.objName
        obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS;
        send chan_U_NS( mes); receive chan_U_CS( mes);
        if mes.type ≠ UNKNOWN →
          if addrHost = mes. addr →              # obiect local
                obtine o conexiune chan_U_CS, chan_U_OBJ de la chanu_OBJ pentru mes.objName;
                send chan_U_OBJ(dt); receive chan_U_CS(dt); send chan_R(dt);
          [] addrHost ≠ mes. addr →              # obiect la distanta
                incearca pina se reuseste stabilirea unei conexiuni chan_KT_CS, chan_KO_CS
                                                        de la CS-ul din nodul mes.addr
                send chan_KT_CS (dt); receive chan_KO_CS(dt); send chan_R(dt);
          fi
        fi
return
Services(typ:int, data: BT_ACK_PACKET)::
var pA: BT_ACK_PACKET; pckt: PACKET; aut: RD_PACKET;          # variabile de lucru
var errCode: int, slot: int, res: ACK_ID_TRANS, waiting:bool  # variabile de lucru
var bto: BT_OBJECT;
        if typ = LOCAL →                           # cerere de la o aplicatie locala
          copiaza data la pA; aplName:= data.aplName; semId:= data.semId; shmId:= data.shmId;
          if nu se poate realiza atasarea la memoria partajata → return
          aut.type:=GET_TRANS_ID; aut.pri:= pA.pri;
          obtine o conexiune chan_K_CS, chan_K_CM de la chans_CM; send chan_K_CM(aut);
          if (errCode:= receive chan_K_CM( res))=TIME_OUT→
                pckt:.type := INTERRUPT; aloca slot pt pckt; send chan_S_TM(slot); exit;fi
          [] errCode=SUCCES →
                pckt.type :=ACK_TM_BEGIN_TRANSACTION; pckt.pri:= res.pri;pckt.errCode := SUCCES
                pckt.transId := res.trans.Id;aloca slot pt pckt; send chan_S_TM(slot)
          fi
          waiting:= true
          do waiting = true →
                receive chan_S_CS(slot);
                TreatMessage(slot, LOCAL);        # trateaza un mesaj de la o aplicatie locala
```


BUPT

```
            od
        [] typ = DISTANCE →
            if nu se poate crea si atasa la o zona de mem. partajata shmId sau semaf. semID  return
            completeaza cimpurile bto cu: .type:=BEGIN_TRANSACTION, .transId:=data.transId,
                    .shmId:=data.shmId, .semId:=data.semId, .semNo:=FIRST_UNUSED, .queueNo:=0
                    .aplName:= data.aplName
            obtine o conexiune chan_U_CS, chan_U_OBJ de la chanu_OBJ pentru data.objName;
            send chanu_U_OBJ(bto);  waiting:= true
            do waiting = true →
                    instaleaza un handler pentru SIGALRM
                    programeaza un semnal de alarma pentru TIME_OUT_REQ
                    if ((errCode: = receive chan_K_CS(pckt) = SUCCESS)→
                        TreatMessage(slot, DISTANCE);
                        if slotsTab[slot].type = ACK_ABORT or slotsTab[slot].type = ACK_COMMIT→
                                waiting = true; elibereaza zona de memorie partajata
                        fi
                    [] errCode ≠ SUCCESS →
                                genereaza un slot cu slotsTab[slot]=INTERRUPT;
                                send chan_S_OBJ(slot)
                                again: receive chan_S_CS(pckt);i
                                if pckt.type ≠  ACK_INTERRUPT→ goto again
                                elibereaza zona de memorie partajata si semafoarele; exit
                    fi
            od
        fi
return
TreatMessage(slot: int, where:int)::
var bto: BT_OBJECT; mes: RD_ OBJECT   ; wp:PACKET
        pckt:= ref(slotsTab[slot]);
        if where=LOCAL  →                               # mesaj de TM local
                if pckt.type  =  FIRST_REQUEST  →       # intii determina localizarea obiectului
                completeaza mes cu valorile .type:= GET_OBJECT, .objName:= pckt.objName
                obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS;
                send chan_U_NS( mes);  receive chan_U_CS( mes);
                if mes.type = UNKNOWN  →
                        transErr:= true; objTab[pckt.objId].name:= ""
                        completeaza slot cu cimpurile .type:= ACK_FIRST_REQUEST
                        .errCode:= EOBJ_NOT_FOUND; transId:= -1;send chan_S_TM(slot); return
                [] if  mes.type = OBJ_FOUND  →
                        objTab[pckt.objId].name:= pckt.objName;             # obiect local
                        if addrHost = mes. host →
                                completeaza cimpurile bto cu:
                                .type:=BEGIN_TRANSACTION, .transId:=pckt.transId,
                                .shmId:=shmId, .semId:=semId,
                                .semNo:= semList[semLIstIdx];semLIstIdx:=semLIstIdx+1
                                .queueNo:=semNo-FIRST_UNUSED;  .aplName:= aplName
                                obtine o conexiune chan_U_CS, chan_U_OBJ
                                                        de la chanu_OBJ pentru mes.objName;
                                send chanu_U_OBJ(bto);  waiting:= true
                                send  chan_S_OBJ[semNo](pckt);
                                objTab[pckt.objId].sem:=  bto.semNo; objTab[pck.objId].local:=1
                        [] addrHost ≠ mes. host                     # obiect la distanta
                            objTab[pck.objId].local:=0;
                            if dupa MAX_TRY  nu se poata obtine o conex. cu chans_CS  de adr. mes.addr→
                                transErr:= true; objTab[pckt.objId].name:= ""
                                completeaza slot cu cimpurile .type:= ACK_FIRST_REQUEST
                                .errCode:= EOBJ_NOT_FOUND; transId:= -1;send chan_S_TM(slot);
                                return
                            [] se poate obtine o conex. chan_KT_CS,chan_KO_CS
                                objTab[pckt.objId].socket:= chan_KD_CS
```


BUPT

```
                                        mes.type:= NTEWORK_BEGIN_TRANSACTION;
                                        mes.transId := pckt.transId; mes.pri:= pckt:.pri;
                                        errCode := send chan_KO_CS (bto)
                                        if errCode ≠ SUCCESS →
                                            pck.type:= INTERRUPT; send chan_S_TM(slot);
                                            return
                                        [] errCode = SUCCESS
                                            creaza un proces fiu pid pentru obiectul la distanta din tranz. locala
                                               inchide toate canalel dinamice cu exceptia chan_KD_CS
                                               instaleaza o functie pentru tratarea semnaluli SIGALRM
                                               do true
                                                  programeaza un semnal de alarma pentru
                                                  errCode:= receive chan_KT_CS(wp);
                                                  if errCode=SUCCES →
                                                     aloca slot; slotsTab[slot]:=wpckt;
                                                     send chan_S_TM(slot);
                                                     if w.type=ACK_ABORT or w.type = ACK_COMMIT→
                                                          exit
                                                  [] errCode ≠SUCCES →
                                                          pck.type:= INTERRUPT; send chan_S_TM(slot);
                                                  inchide canalele; exit
                                                  fi
                                               od
                                            pidTab[pckt.objId[:=pid
                                        fi
                                   fi
                          fi
                 [] pckt.type = REQUEST or pckt.type = PREPARE_TO_COMMIT or pckt.type = COMMIT→
                          if transErr=false → Transmit(slot, false)
                 [] pckt.type = ABORT→ Transmit(slot, false)
                 [] pckt.type = INTERRUPT_ABORT → pckt:= ABORT;Transmit(slot, true)
                 [] pckt.type = INTERRUPT_COMMIT → pckt:= COMMIT;Transmit(slot,true)
                 [] pckt.type = ACK_FIRST_REQUEST →
                       if objTab[pckt.objId].local ≠0 →
                        if pckt.transId=-1 →semList[semListIdx]:= objTab[pckt.objId].sem; semListIdx:=semListIdx-1
                              fi
                       fi
                       send chan_S_MT(pckt)
                 [] pckt.type = ACK_REQUEST or pckt.type = PREPARE_TO_COMMIT
                              or pckt.type =ACK_COMMIT or pckt.type =ACK_ABORT →
                       send chan_S_MT(pckt)
                 [] pckt.type = END_TRANSACTION →
                       pckt.type := CS_END; send chan_S_MT(pckt); waiting := false;
                 fi
           [] where = DISTANCE →
                 if pckt.type = FIRST_REQUEST or pckt.type=PREPARE_TO_COMMIT or
                       pckt.type = ABORT or pckt.type = COMMIT→
                       send chan_S_OBJ(); receive chan_S_CS(slot)
                       errCode:=send chan_KT_CS (slotsTab[slot])
                       if errCode ≠ SUCCESS →
                              rezerva un slot; slotsTab[slot].type:= INTERRUPT; send chan_S_OBJ(slot);
                              again: receive chan_S_CS(slot);
                              if slotsTab[slot].type ≠ INTERRUPT → goto again fi
                              elibereaza zona de memorie partajata
                       fi
           fi
return
Transmit(slot:int, abort: bool):: # se apeleaza pentru a transmite comenzile locale tranzactiei la distanta
var bto: BT_OBJECT, mes: RD_ OBJECT, sem: int,          # variabile de lucru
        pckt:= ref(slotsTab[slot]);
```


BUPT

```
        if objTab[pckt.objId].objName = ∅  →                # obiect cu nume nul
           if pckt.type = ABORT →
              genereaza un slot cu .type:=ACK_ABORT, .objName:=pckt.nameObj; .objId:=pckt.objId
              send chan_S_TM(slot)
              return
           fi
        fi
        if objTab[pckt.objId].local = 0  →          # obiect local, pe acelasi nod cu tranzáctia
         sem:= objTab[pckt.objId].sem; send chan_S_OBJ(pckt) #transmite pachetul la ob. local prin SHM
        [] if abort →                                 # situatia cind nodul obiectului nu mai raspunde
           inchide canalul vechi objTab[pckt.objId].chan
           distruge eventualul proces care mai asteapta la receive pt. ca CS-ul pereche sa forteze INTERRUPT
                                         identificatorul procesului este memorat in pidTab[pckt.objId]
           bto.type:=pckt.type; bto.transId:=pckt.transId;
           creaza un proces fiu care executa:
              inchide toate canalele din objTab[].chan
              completeaza mes cu valorile .type:= GET_OBJECT, .objName:= pckt.objName
              obtine o conexiune chan_U_CS, chan_U_NS de la chanu_NS;  # intii determina
              send chan_U_NS(mes); receive chan_U_CS(mes);            # localizarea obiectului
              instaleaza cite handler pentru tratare SIG_PIPE
              incearca  periodic, la infinit, obtinerea unei conexiuni
                        chan_KT_CS, chan_KO_CS,   cu CS-ul mes.host
              if conexiunea s-a putut stabili →
                       send chan_KO_CS(bto); receive chan_KT_CS( bto);
                       pckt.type:=bto.type; send chan_S_TM(slot);
              fi
           fi
        [] abort = false →
           creaza un proces fiu care executa: # este necesar un fiu pentru cazul cind nodul  obi. nu mai rasp.
              send  objTab[pckt.objId].chan(pckt)  # se transmite  SIGPIPE si este distrus numai acest fiu
        fi
return
```

9.4.5. Proiectarea serverelor de nume

Rolul serverelor de nume este de a menține o listă cu identitatea obiectelor conectate în sistem și localizarea lor. Un server de nume este contactat de un server de comunicații în trei situații:

(1) cînd se apelează prima operație asupra unui obiect nou implicat în tranzacție, pentru ca serverul de comunicații să cunoască unde va plasa cererea (ca și următoarele pentru același obiect): local, în memoria partajată sau contactînd serverul de comunicații de pe nodul obiectului;

(2) la conectarea sau eliminarea unor obiecte în sistem, cînd managerul central transmite modificarea efectuată;

(3) la conectarea nodului în sistem, cînd managerul central transmite identitatea și localizarea tuturor obiectelor integrate pînă în acel moment în sistem.

Managerul central transmite informații serverelor de nume via serverele de comunicații.

Descrierea SR a serverelor de nume este:

```
NameServer()::
var pcktRD, obj: RD_PACKET
var  res:ACK_ID_TRANS, mes:BT_OBJECT
var hostAddr: ADDR
   #faza de initializare
   initializeaza lista  obiectelor conectate ca vida
   determina adresa nodului curent la hostAddr
   #faza de primire si tratare cereri
   do true →
       accepta o conexiune  chan_U_S, chan_U_NM pe canalul chanu_NS
       receive  chan_U_(typ)
       if typ = RECORD →
             receive chan_U_NS(pcktRD); inscrie noul obiect in lista obiectelor conectate
             if  pcktRD.host = hostAddr  →
                   mes. type := ACK_RECORD; send  chan_U_S(mes);
```


BUPT

```
        [] typ = DELETE →
                receive chan_U_NS(pcktCD); sterge obiectul din lista obiectelor conectate
        [] typ = RECORD_END →
                receive chan_U_NS(pcktCD); send chan_U_S(pcktCD);
        [] typ = GET_OBJECT →
                receive chan_U_S(pcktRD);
                if pcktRD.objName exista in lista de obiecte obj →
                        pcktRD.type:= OBJ_FOUND; pcktRD.host:=obj.host;pcktRD.port:=obj.port;
                [] pcktRD nu exista in lista de obiecte obj →
                        pcktRD.type:= UNKNOWN;
                send chan_U_S(pcktRD);
                fi
        fi
    od
```

9.4.6. Proiectarea managerului central

Managerul central a fost introdus în vederea realizării urmatoarelor două scopuri: (1) ținerea (centralizată) a evidenței nodurilor conectate la sistem; (2) livrarea unui număr de prioritate unic tranzacțiilor inițiate în mediul distribuit.

Descrierea SR a managerului central este:

```
CentralManager()::
var pcktRD, obj: RD_PACKET
var pcktCD, node: CD_PACKET
var res:ACK_ID_TRANS
  #faza de initialiazare
  if nu se poate deschide fisierul TrIdPr cu val. curenta de acordat id. de tranzactii si prioritatii → Panic()
  initializeaza lista de noduri conectate si a obiectelor conectate ca vide
  #faza de primire si tratare cereri
  do true →
        accepta o conexiune chan_K1_CS, chan_K1_CM pe canalul chans_CM
        receive chan_K_CM(typ)
        if typ = CONNECT →
                receive chan_K_CM(pcktCD); inscrie noul nod in lista nodurilor conectate
                fa obj din lista de obiecte introduse in sistem →
                        completeaza la pcktRD cimpurile .host:= obj.host, .type:=RECORD,
                                                        .objNam:=obj.objName, .port:=obj.port
                        send chan_K1_CS(pckRD);
                af
                pckt.type:=RECORD_END ; send chan_K1_CS(pckRD);
        [] typ = DISCONNECT →
                receive chan_K_CM(pcktCD); sterge nodul din lista nodurilor conectate
        [] typ = RECORD →
                receive chan_K_CM(pcktRD); inscrie noul obiect in lista obiectelor conectate
                fa node din lista de noduri introduse in sistem →
                        obtine o conexiune chan_K2_CS, chan_K2_CM de la canalul chans_CS
                                                        de pe nodul node.host
                        send chan_K2_CS(pckRD);
                af
                inchide canalele chan_K2_CS, chan_K2_CM
        [] typ = DELETE →
                receive chan_K_CM(pcktRD); sterge obiectul in lista obiectelor conectate
                fa node din lista de noduri introduse in sistem →
                        obtine o conexiune chan_K2_CS, chan_K2_CM de la canalul chans_CS
                                                        de pe nodul node.host
                        send chan_K2_CS(pckRD);
                af
                inchide canalele chan_K2_CS, chan_K2_CM
        [] typ = GET_TRANS_ID →
                receive chan_K_CM(pcktRD);
                citeste si apoi incrementeaza transId din fisierul local TrIdPri;
```


BUPT

```
                rescrie valoarea incremenatat in fisier ; res.transId:=transId;
            if pcktRD.pri = -1 →
                citeste si apoi incrementeaza pri din fisierul local TrIdPri;
                rescrie valoarea incremenatat in fisier ; res.pri:=pri;
            [] pcktRD.pri ≠ -1 → res.pri:=pcktRD.pri
            res.type:= TRANS_ID_ACK; send chan_K_CS(res)
            fi
        fi
    od
```

9.4.7. Proiectarea managerilor de obiecte

S-a arătat la 9.3.6. că managerii de obiecte încapsulează două tipuri de date (obiectele şi informaţia de sincronizare) şi patru tipuri de funcţii care, în principal, tratează lucrul cu obiecte, sincronizarea accesului la obiecte, recuperarea obiectelor după defecte şi integrarea obiectelor în sistem. Pînă în momentul de faţă, în DOSTP au fost implementaţi manageri de obiecte care folosesc o schemă de sincronizare optimistă şi o tehnică de recuperare redo-only (adecvată pentru schemele optimiste).

Implementarea managerilor se face folosind tehnica programării pe obiecte (vezi 10.3.).

În cele ce urmează, se prezintă descrierea SR a unui manager de obiecte, fără a se lua în considerare modul de implementare obiectual.

```
ObjectManager()::
type ACLOG_TYPE = record of                      # structura articoleleor din fisierul cu tranzactiile care
        (transId: int,                           # au executat prepare: identificatorul tranzactiei
        pri: int,                                # prioritatea tranzactiei
        op: int,                                 #  operatia de executat dupa prepare: commit sau abort
        logPC: int)                              # deplasamentul in fisierul PCLOG
type TSFILE_TYPE = record of                     #  structura articoleleor din fisierul TSFILE care retine
        (transId: int,                           #  tranzactiile terminate: identificatorul tranzactiei
        pri: int)                                # prioritatea tranzactiei
var: serverName[1:MAX_APL_LEN], hostName[1:MAX_APL_LEN];   # memoreaza nume
var: aut: RD_PACKET                              # folosit la conectarea obiectului
var mes:BT_OBJECT, pckt:PACKET                   # mesaje receptionate si trimise
var ts: int, pri:int, op:int, logPC:int          # informatii din ACLOG pentru o tranzactie
var semId:int, semNo:int, queueNo:int, shmId:int # informatii necesare pentru accesul la SHM
var waiting: bool:=true, conn :int:= false       # indicatori

    # faza de initializare
    if  nu se poate crea si deschide canalul de comunicatie propriu chan_OBJ → Panic() fi
    obtine numele nodului curent la hostName si numele obiectului serverName
    completeaza aut cu .type:=RECORD, .objName:= serverName; .host:=hostName
    if  nu se poate crea si deschide canalul de comunicatie cu chanf_CS → Panic() fi
    send chanf_CS(aut);
    if nu se poate deschide fisierul ACLOG in modul append → Panic() fi
    if nu se poate deschide fisierul PCLOG in modul append → Panic() fi
    if nu se poate deschide fisierul TSFILE in modul append → Panic() fi
    seteaza modul de lucru fara buferare pentru fisierele ACLOG, PCLOG, TSFILE
    if nu se poate deschide si mapa in memorie fisierul cu copia obiectului → Panic() fi
    if nu se poate deschide si mapa in memorie fisierul cu informatiile de stare ale tranzactiilor → Panic() fi
    if nu se poate crea semaforul semObj pentru accesul la copia partajata a obiectului in memorie → Panic() fi
    if nu se poate crea semaforul semTs pentru accesul la fisierul de tranzactii TSFILE → Panic() fi
    # faza de recuperare initiala a obiectelor
    pozitionare la inceputul fisierului ACLOG
    fa articolele fisierului ACLOG →
        citeste cimpurile unui articol ACLOG_TYPE la ts, pri, op, logPC
        if ts>0 Recovery( ts, pri, op, logPC) fi        # realizeaza commit sau abort specific obiectului respectiv
    af
    creaza o alta copie pentru fisierul PCLOG avind inregistrarile invalidate eliminate
    # reface efectele efecte tranzactiilor ramase inca neterminate
    pozitionare la inceputul fisierului PCLOG
    fa articolele fisierului PCLOG →
```


BUPT

```
        citeste cimpurile unui articol la ts, pri, op, ind, val
        completeaza oldTrans[oldTransIdx]: .transNo:=ts, .pri:= pri, logPC:= pozitia curenta din PCLOG
        seteaza lock-urile pentru operatia op, ind respectiva
  af
# faza de asteptare a unui mesaj pentru implicarea intr-o intr-o tranzactie sau de terminare tranzactie
do true →
   stabileste o conexiune chan_U_OBJ, chan_U_CS la canalul chanu_OBJ
   receive chan_U_OBJ (mes);
   if conn = false →
      if mes.type = ACK_RECORD and mes.errCode = SUCCESS → conn:= true;
   []conn = true →
      if mes.type = TRANSACTION_ACCEPTED →
         insereaza tranzactia in lista de tranzactii active
      [] mes.type = TRANSACTION_ACCEPTED →
         insereaza tranzactia in lista de tranzactii active
      [] mes.type = TRANSACTION_ENDED →
         sterge tranzactia in lista de tranzactii active
      [] mes.type = TRANSACTION_INTERRUPTED →
         sterge tranzactia in lista de tranzactii active
      [] mes.type = BEGIN_TRANSACTION →
         ts:= mes.transId;
        creaza un proces fiu care executa:
          ts:= mes.transId; pr:=mes.pri, semid:=mes.semId; shmId:=mes.shmId;
          queueNo:=mes.queueNo; semNo:= mes.semNo;
          if nu se poate realiza atasarea la zona de memorie partajata →
                mes.type:= TRANSACTION_NOT_ACCEPTED; send chanu_OBJ(mes);exit
          fi
          waiting:= true
          mes.type:= TRANSACTION_ACCEPTED; send chanu_OBJ(mes)
          do waiting = true →
                receive chan_S_OBJ(pckt);
                if pckt.type = FIRST_REQUEST→
                        trateaza cererea seteaza pckt.errCode;
                        pckt.type := ACK_FIRST_REQUEST;send chan_S_CS(pckt)
                [] pckt.type = REQUEST →
                        trateaza cererea seteaza pckt.errCode;
                        pckt.type := ACK_REQUEST;send chan_S_CS(pckt)
                [] pckt.type =PREPARE_TO COMMIT →
                        trateaza cererea si seteaza pckt.errCode;
                        pckt.type := ACK_PREPARE_TO_COMMIT; send chan_S_CS(pckt)
                [] pckt.type = COMMIT →
                        trateaza cererea si seteaza pckt.errCode;
                        pckt.type := ACK_COMMIT or send chan_S_CS(pckt);
                        mes.type:= TRANSACTION_ENDED; send chanu_OBJ(mes); waiting := false
                [] pckt.type = ABORT →
                        trateaza cererea si seteaza pckt.errCode;
                        pckt.type := ACK_ABORT  send chan_S_CS(pckt);
                        mes.type:= TRANSACTION_ENDED; send chanu_OBJ(mes); waiting := false
                [] pckt.type = INTERRUPT →
                        trateaza cererea si seteaza pckt.errCode; send chan_S_CS(pckt);
                        mes.type:= TRANSACTION_INTERRUPTED; send chanu_OBJ(mes);
                        pckt.type:= ACK_INTERRUPT;send chan_S_CS(pckt);waiting := false;
                fi
          elibereaza memoria partajata
          exit
      [] mes.type = ABORT or mes.type = COMMIT →
         creaza un proces fiu care executa:
                mes.transId:=trans.Id; op:=mes.type;
                if transId exista in fisierul TSFILE → errCode := RECOVERY_OK # tranzactie terminata
```


BUPT

```
                [] transId nu exista in fisierul TSFILE
                  do cit timp transId exista exista in tabloul tranzactiilor in curs de recuperare→sleep(TIME)
                   if starea tranzactiei din oldTrans = WORKING →
                          mes.errCode := RECOVERING      # recuperare deja in lucru pentru alt proces
                   [] starea tranzactiei din oldTrans =  DONE →
                          mes.errCode := DONE                # recuperare rezolvata
                   [] starea tranzactiei din oldTrans =  UNCHANGED →
                          inregistreaza in oldTrans starea tranzactiei transId ca  RECOVERING
                          Recovery( ts, pri, op, logPC);  inregistreaza in oldTrans starea tr. transId ca  DONE
                          mes.errCode := RECOVERY_OK
                  [] tranzactia nu exista in oldTrans →
                          mes.errCode:= TRANS_NOT_FOUND
                  fi
                  if errCode = RECOVERY_OK→
                          if op = ABORT mes.type := ACK_ABORT
                          [] op = COMMIT mes.type := ACK_COMMIT
                          fi
                  fi
                  send  chan_U_CS_OBJ(mes)
                  exit
                od
      fi
    fi
  od
```


BUPT

Capitolul 10.

IMPLEMENTAREA SISTEMULUI DOSTP ÎN REȚELE UNIX ȘI MICROSOFT WINDOWS (WIN32)

Capitolul de față arată cum se implementează sistemul DOSTP, a cărei proiectare a fost prezentată în capitolul 9, în retele Unix și Microsoft Windows (Win 32).

10.1. Particularități de implementare

Subcapitolul 10.1 se referă la implementarea tuturor componentelor sistemului DOSTP, cu excepția managerilor de obiecte, a căror implementare este prezentată (pentru un tip de date particular) în subcapitolul 10.3.

10.1.1. Particularități de implementare în rețele Unix

Implementarea sistemului DOSTP în rețelele Unix s-a făcut urmărind îndeaproape proiectarea prezentată în descriere SR în capitolul 9. Se fac însă următoarele observații:

(1) Canalele de comunicație între serverele de comunicație ale nodurilor, pe de o parte, și, pe de altă parte între serverele de comunicație și serverul central au fost implementate prin socluri Inet. Pentru recepționarea și emisia mesajelor din/pe aceste canale au fost scrise două funcții speciale care, apelînd (eventual de mai multe ori) funcțiile de sistem *read*() și *write*() asigură citirea/scrierea întregului mesaj (un anumit numar de octeți). Acest lucru este necesar ținînd cont de specificul soclurilor TCP (fluxuri de octeți).

(2) Canalele de comunicație statice server de nume -> server de comunicație, aplicație -> manager de tranzacții, manager de tranzacții -> server de comunicații și manager de obiecte -> server de comunicație s-au implementat prin canale FIFO, iar canalele de comunicație server de comunicație -> manageri de obiecte, server de comunicație -> server de nume s-au implementat prin socluri Unix.

(3) Comunicația locală dintre aplicație <-> manager de tranzacții, manager de tranzacții <-> server de comunicații, server de comunicații <-> manager de obiecte se realizează printr-o zonă de memorie partajată, a cărei creare se face în managerul de tranzacții astfel:

```
if ((shmId=shmget(IPC_PRIVATE, DIM_SHM, IPC_CREAT | S_IRUSR | S_IWUSR)) < 0)
                Panic ("TM->Error : Cannot creat shared memory");
if (pShm = (SHM*) shmat (shmId, 0, 0)) = (SHM*)-1)
                Panic ("TM->Error : Cannot attach to shared memory\n");
```

(4) Pentru accesul la zona de memorie partajată se folosesc semafoarele *S_APL* (de aplicație), *S_TM* (de managerul de tranzacții), *S_CS* (de serverul de comunicații), în scopul recepționării pachetelor (emițătorul execută Up(semafor), iar receptorul Down(semafor)). În plus, mai sunt prevazute semafoarele: *S_SHM* pentru accesul exclusiv la lista de pachete libere din zona de memorie partajată (în momentul alocării sau eliberării unui pachet) și *S_FREE_SLOT* inițializat la numărul total de pachete disponibile în zona de memorie partajată, decrementat (Down(semafor)) la fiecare alocare a unui pachet și incrementat la fiecare eliberare de pachet (astfel încît în situația că nu mai sunt pachete libere în zona, apelantul este blocat).

Crearea și inițializarea semafoarelor se face imediat după crearea zonei de memorie partajată astfel:

```
if ((semId=semget(IPC_PRIVATE, SEM_NO, IPC_CREAT | S_IRUSR | S_IWUSR)) < 0)
                Panic ("TM->Error : Cannot create semaphores");
if (semctl (semId, S_SHM, SETVAL, 1) == -1 ||
        semctl (semId, S_APL, SETVAL, 0) == -1 ||
        semctl (semId, S_TM, SETVAL, 0) == -1 ||
        semctl (semId, S_CS, SETVAL, 0) == -1 ||
        semctl (semId, S_FREE_SLOT, SETVAL, MAX_SLOTS) == -1)
  )
        Panic ("TM -> Cannot initialize semaphores\n");
```

(5) Comunicația între două procese prin zona partajată a fost reprezentată în descrierea SR din subcapitolul 9.4, sub forma unor operații de recepție și emisie. Implementarea comunicației prin zona de

BUPT

memorie partajată, de exemplu, în cazul managerului de tranzacţii care poate trimite pachete atît aplicaţiei cît şi serverului de comunicaţii a fost realizată prin următoarea funcţie:

```
void SendPacket (int dest, int slot) {
        WaitOnSemaphore (semId, S_SHM);
        switch (dest)
        {
        case APL:       aplList[aplListIdx] = slot;   /*se depune numarul pachetului care va fi transmis */
                        aplListIdx=(aplListIdx+1) % MAX_SLOTS;   /*înca un pachet depus în coada */
                        signalSemaphore(semId,S_APL);            /*se anunta aplicatia */
                        break;
        case CS:        commServList[commServListIdx]=slot;
                        commServListIdx=(commServListIdx+1) % MAX_SLOTS;
                        SignalSemaphore(semId, S_CS);
                        break;
        default:        Debug("No such destination\n");
        }
        SignalSemaphore(semId, S_SHM);
}
```

(6) Rezervarea şi dealocarea unui pachet din zona de memorie partajată a fost implementată astfel:

```
void FreeSlot(int slot) {
        WaitOnSemaphore(semId, S_SHM);          /* acces exclusiv la zona partajata */
        freeSlotList[--freeSlotListIdx]=slot;   /*pachetele eliberate se depun la stinga indexului */
        SignalSemapfore(semId, S_SHM);          /* s-a eliberat zona */
        SignalSemaphore(semId, S_SLOT_FREE);    /* s-a mai eliberat un pachet */
}
int GetSlot(void) {
        int slot;
        WaitOnSemaphore(semId, S_SLOT_FREE);/*se asteapta eliberarea unui pachet */
        WaitOnSemaphore(semId, S_SHM);          /* acces exclusiv la zona partajata */
        slot=freeSlotList[freeSlotListIdx++];   /*se ocupa un pachet */
        SignalSemaphore(semId, S_SHM);          /* se elibereaza zona */
        return slot;
}
```

(7) Managerii de tranzacţii şi serverele de comunicaţii dedicaţi unei tranzacţii au fost implementaţi în procese separate create de managerul de tranzacţii, respectiv serverul de comunicaţii. Aceştia există numai pe parcursul desfăşurării unei tranzacţii. Pentru a nu se genera procese *zombie* la terminarea proceselor fii, a trebuit captat şi tratat semnalul SIGCHLD.

(8) Managerul de tranzacţii, serverul de comunicaţii, serverul de nume şi managerul central au fost realizaţi ca programe XWindows. Acestia prezintă o interfaţă simplă care oferă şi facilităţi pentru vizualizarea tranzacţiilor în lucru (managerul de tranzacţii), a obiectelor conectate la sistem (serverul de nume şi managerul central) şi a nodurilor conectate la sistem (managerul central). Recepţionarea mesajelor pe canalele statice se face asincron, în cadrul unor funcţii declarate *XtInputCallbackProc*.

(9) Pentru ca semnalele Unix, care sosesc asincron, să nu întrerupă un apel sistem *semop* ce punea în aşteptare un proces la un semafor, implementarea unei funcţii *WaitOnSemaphore()* trebuie facută avîndu-se în vedere acest lucru, astfel:

```
void WaitOnSemaphore(int semId, int sem) {
        struct sembuf semBuf;
        int code;
        semBuf.sem_num=sem;
        semBuf.sem_op=-1;
        semBuf.sem_flg=0;
        again:  errno=0;
        code=semop(semId, &semBuf, 1);
        if (code==-1) {
                if (errno==EINTR) goto again;
                if (errno==EIDRM || errno==EINVAL)
                                Panic(" Access to semaphores lost\n");
        }
}
```

(10) Pentru detectarea pierderii legăturii cu un nod se captează şi tratează semnalele SIGALARM şi SIGPIPE.

BUPT

10.1.2. Particularități de implementare în rețele Microsoft Windows (Win 32)

Pentru varianta Windows au fost implementate componentele DOSTP care trebuie să gestioneze un nod ce conține obiecte: serverul de comunicație, serverul de nume și managerii de obiecte (pentru managerii de obiecte vezi subcapitolul 10.3).

Ca și în varianta Unix, implementarea serverului de comunicație și a serverului de nume s-a facut urmărind proiectarea prezentată la 9.4. Trebuie facute însă următoarele observații:

(1) Canalele de comunicație server de comunicație <-> manageri de obiecte, server de comunicație <-> server de nume au fost implementate folosind numai socluri Inet. Nu se mai pot folosi, ca în varianta Unix socluri locale (specifice numai pentru Unix) sau canale FIFO.

(2) Comunicația locala, dintre serverul de comunicație și managerul de obiecte se face tot printr-o zonă de memorie partajată. Crearea ei se face astfel:

```
if (( hMmf = CreateFileMapping ((HANDLE)0xffffffff, NULL, PAGE_READWRITE, 0, sizeof(structSHM), szBuff))
                                                        ==NULL)
        Panic ("CS->Error : Cannot creat shared memory"); ");
if ((*ppShm = (struct SHM*)MapViewOfFile( hMmf, FILE_MAP_WRITE, 0, 0, sizeof(struct SHM))) == NULL )
        Panic ("CS->Error : Cannot map shared memory");
```

(3) Pentru accesul la zona de memorie partajată se folosesc analog semafoarele descrise la punctul (4) de la 10.1.1 dar asocierea lor, în mod unic, la o zonă de memorie partajată se face asigurîndu-le un nume unic în sistem astfel:

```
  sprintf( szBuff, "SHM%s%d", objName, transId);
  if ( NULL == (*phSemSHM = CreateSemaphore( NULL, 1, 1, szBuff)))
        Panic ("CS->Error : Cannot create SHM semaphore");

  sprintf( szBuff, "CS%s%d", objName, transId);
  if ( NULL == (*phSemCS = CreateSemaphore( NULL, 0, NR_MAX_SLOTURI, szBuff)))
        Panic ("CS->Error : Cannot create CS semaphore");

  sprintf( szBuff, "SLOT%s%d", objName, transId);
  if ( NULL == (*phSemSlot = CreateSemaphore( NULL, NR_MAX_SLOTURI, NR_MAX_SLOTURI, szBuff)) )
        Panic ("CS->Error : Cannot create SLOT semaphore");
}
```

(4) Rezervarea și dealocarea unui pachet din zona de memorie partajată a fost implementată astfel:

```
void FreeSlot( int slot, HANDLE hSemSHM, HANDLE hSemSlot, struct SHM *pShm)
        WaitForSingleObject( hSemSHM, INFINITE );
        freeSlotList[--freeSlotListIdx] = slot;
        ReleaseSemaphore( hSemSHM, 1, NULL);
        ReleaseSemaphore( hSemSlot, 1, NULL);
}
int GetSlot( HANDLE hSemSHM, HANDLE hSemSlot, struct SHM *pShm, DWORD pid){
        int slot;
        WaitForSingleObject( hSemSlot, INFINITE);
        WaitForSingleObject( hSemSHM, INFINITE);
        slot = freeSlotList[freeSlotListIdx++]
        ReleaseSemaphore( hSemSHM, 1, NULL);
        return slot;
}
```

(5) Comunicația intre două procese prin zona de memorie partajată, se realizează în varianta Windows folosind următoarea funcție SendPacket():

```
void SendPachet(PACHET*pPachet,HANDLE hSemSHM,HANDLE hSem,HANDLE hSemSlot,structSHM*pShm)
{
        int slotNo;
        slotNo = GetSlot( hSemSHM, hSemSlot, pShm, pid);
        WaitForSingleObject( hSemSHM, INFINITE);
        objQueueList[0][objQueueListIdx[0]] = nrSlot;
        objQueueListIdx[0] = objQueueListIdx[0](++indexOb[0]) % MAX_SLOTS;
        ReleaseSemaphore( hSemSHM, 1, NULL );
        memcpy( &slotsTab[ slotNo], pPachet, sizeof( slotsTab[slotNo]));
        ReleaseSemaphore( hSem, 1, NULL);
}
```


BUPT

(6) Serverul de comunicaţii principal şi serverele de comunicaţie care gestionează cîte o tranzacţie sunt implementate în *fire* separate. Fiecare fir este creat şi lansat în execuţie de procesul programului principal la primirea în soclul INET a unui mesaj *RECORD, DELETE, END_RECORD, ABORT, COMMIT, NETWORK_BEGIN_TRANSACTION*, cînd se stabileşte o nouă conexiune. Firul care se creează trebuie să-şi copieze cererea primită şi soclul noii conexiuni, după care procesul programului principal poate trata altă cerere de acceptare a unei conexiuni. Pentru sincronizarea proces principal-fir pe parcursul copierii acestor informaţii se foloseşte un semafor special.

(7) Detectarea pierderii legaturii cu nodul coordonator după stabilirea unei conexiuni se realizează comparînd valoarea returnată de apelurile *recv()* şi *send()* folosite pentru comunicaţie prin socluri cu valoarea SOCKET_ERROR.

10. 2. Exemple de desfăşurare a comunicaţiilor în sistem

Se prezintă în continuare cîteva exemple de desfăşurare a comunicaţiilor în sistem. Sunt luate în considerare toate cele şase situaţii distincte care pot să apară în funcţionarea sistemului DOSTP: (1) conectarea unui nod la sistemul DOSTP; (2) integrarea sau eliminarea unui obiect din sistem; (3) iniţierea unei noi tranzacţii; (4) implicarea unui obiect local şi a unui obiect de la distanţă într-o tranzacţie; (5) realizarea protocolului *two_phase commit*; (6) recuperare după o defecţiune a unui nod şi a reintrării sale în sistem.

Pentru a se deosebi mesajele transmise în canalele create static şi asociate managerilor de tranzacţii, serverelor de comunicaţii, de cele create dinamic, în toate figurile din această secţiune (10.2.) transmiterea mesajelor din prima categorie va fi reprezentată cu linii mai ingroşate.

10.2.1.Conectarea nodurilor şi a managerilor de obiecte la sistemul tranzacţional

Conectarea unui nod la sistemul DOSTP este iniţiată de serverul de comunicaţii. Într-un nod care se conectează la sistemul DOSTP trebuie să se lanseze în execuţie (în această ordine) serverul de comunicaţii, serverul de nume, managerul de tranzacţii şi managerii de obiecte. Înaintea, însă, a conectării oricarui nod trebuie să se lanseze managerul central pe nodul fixat.

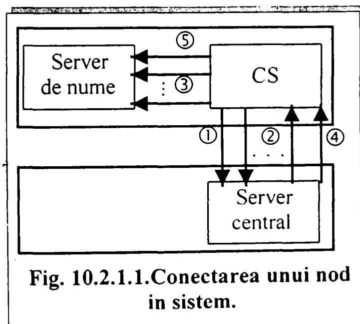


Fig. 10.2.1.1.Conectarea unui nod in sistem.

Circulaţia mesajelor în sistem în cazul conectării unui nod este următoarea (fig. 10.2.1.1):

(1) Serverul de comunicaţii trimite mesajul *CONNECT* managerului central, pe canalul fixat static al acestuia.

(2) Managerul central recepţionează mesajul *CONNECT* şi transmite serverului de comunicaţii (apelant) prin mesaje de tipul *RECORD*, lista cu identificatorii şi localizarea obiectelor conectate la sistem.

(3) Serverul de comunicaţii transmite mai departe mesajele *RECORD* la serverul de nume local.

(4) După transmisia întregii liste a obiectelor, managerul central transmite mesajul *END_RECORD*.

(5) După recepţia mesajului *END_RECORD*, serverul de comunicaţii transmite mesajul mai departe la serverul de nume.

În cazul introducerii sau eliminării unui manager de obiecte la sistem, circulaţia mesajelor este următoarea: (fig. 10.2.1.2.)

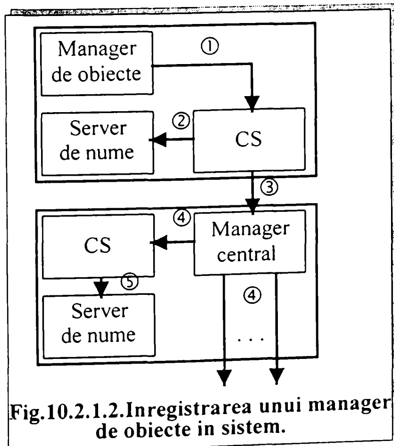


Fig.10.2.1.2.Inregistrarea unui manager de obiecte in sistem.

(1) Managerul de obiecte scrie în canalul (fixat static) al serverului de comunicaţie local (soclu TCP) mesajul pentru conectarea sau eliminarea sa din sistem: *RECORD*, respectiv *DELETE*.

(2) Serverul de comunicaţii trimite mesajul primit (*RECORD* sau *DELETE*) la serverul de nume local.

(3) Serverul de comunicaţii trimite mesajul la managerul central.

(4) Managerul central transmite la serverele de comunicaţie de pe toate nodurile conectate la sistem mesajul de introducere sau eliminare a unui obiect.

(5) Serverele de comunicaţii (cu excepţia celui de pe nodul în care se afla managerul de obiect) transmit mesajul primit la serverele lor de nume locale.

BUPT

10.2.2. Inițierea unei tranzacții

Tranzacțiile se inițiază de către aplicații prin apelul funcției *BeginTransaction()*, care trimite apoi mesajul *BEGIN_TRANSACTION* managerului de tranzacții.

Acțiunile întreprinse de sistemul DOSTP la primirea unui mesaj *BEGIN_TRANSACTION* sunt următoarele (fig. 10.2.2.):

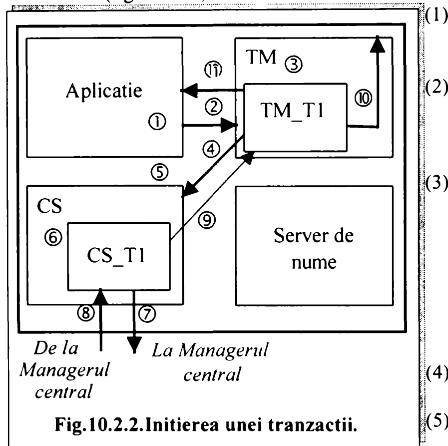


Fig.10.2.2.Initierea unei tranzactii.

(1) Aplicația trimite mesajul *BEGIN_TRANSACTION* în canalul fixat static *chanf_MT* al managerului de tranzacții.

(2) Managerul de tranzacții, care așteaptă în buclă continuă primirea de mesaje în *chanf_TM* recepționează mesajul *APL_BEGIN_TRANSACTION*.

(3) Managerul de tranzacții (TM) creează un proces fiu, TM_T1 care va deservi toate cererile trimise de tranzacția T1 în memoria partajată; TM transmite mesajul *APL_BEGIN_TRANSACTION* fiului nou creat; TM_T1 va crea și se va atașa la o zona de memorie partajată în care se vor transmite, local, toate mesajele pentru acea tranzacție.

(4) TM_T1 transmite serverului de comunicații mesajul *TM_BEGIN_TRANSACTION*.

(5) Serverul de comunicații (CS) recepționează mesajul *TM_BEGIN_TRANSACTION*.

(6) Serverul de comunicații, CS, creează un fiu CS_T1 care va gestiona exclusiv comunicațiile pentru tranzacția respectiva T1.

(7) CS_T1 transmite un mesaj *GET_TRANS_ID* la managerul central, pentru a obține un identificator și o prioritate pentru tranzacție.

(8) Managerul central recepționează mesajul *GET_TRANS_ID*, alocă un identificator și o prioritate pentru tranzacție și răspunde apelantului transmițîndu-i identificatorul și prioritatea tranzacției.

(9) CS_T1 recepționează mesajul transmis de managerul central și transmite managerului TM_T1 mesajul *ACK_TM_BEGIN_TRANSACTION*

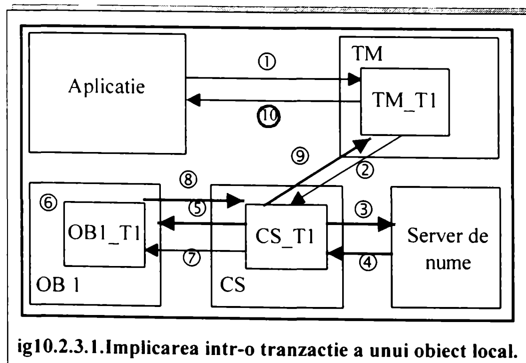
(10) TM_T1 transmite managerului de tranzacții TM mesajul *TRANSACTION_ACCEPTED*.

(11) TM_T1 transmite aplicației *ACK_APL_BEGIN_TRANSACTON*.

10.2.3. Exemple de implicare a obiectelor în tranzacții

Există două situații distincte de implicare a obiectelor în tranzacții: (1) obiectul pentru care s-a cerut operația (prima operație pentru acel obiect) este local; (2) obiectul este situat la distanță.

Actiunile întreprinse de sistem pentru implicarea unui obiect local într-o tranzacție T1 sunt următoarele (fig. 10.2.3.1):



ig10.2.3.1.Implicarea intr-o tranzactie a unui obiect local.

(1) aplicația trimite mesajul *REQUEST* (în zona de memorie partajată), mesaj care cuprinde operația ceruta.

BUPT

(2) managerul de tranzacţii TM_T1 recepţionează cererea şi, în urma analizei tabelelor sale, constatînd că obiectul nu a mai fost apelat în cadrul tranzacţiei T1, modifică tipul mesajului primit la *FIRST_REQUEST* şi îl trimite mai departe la CS_T1.
(3) CS_T1 recepţionează mesajul pentru a localiza obiectul, contactează serverul de nume transmiţîndu-i un mesaj de tipul *GET_OBJECT*.
(4) serverul de nume recepţionează mesajul şi trimite răspunsul la CS_T1.
(5) CS_T1 trimite mesajul *BEGIN_TRANSACTION* în canalul fixat static al managerului obiectului.
(6) managerul obiectului crează un fiu care se va ocupa de tranzacţia respectivă şi care se ataşează la zona de memorie partajată (OB_T1) şi aşteaptă primirea de mesaje în zona de memorie partajată.
(7) CS_T1 trimite în zona de memorie partajată pentru OB_T1 cererea primită în mesajul *FIRST_REQUEST* de la TM_T1.
(8) OB_T1 primeşte mesajul şi trimite catre CS_T1 răspunsul la cerere într-un mesaj de tipul *ACK_FIRST_REQUEST*.
(9) CS_T1 primeşte mesajul şi transmite răspunsul la TM_T1.
(10) TM_T1 transmite răspunsul cererii la aplicaţie.

În scopul implicării unui obiect de la distanţă, acţiunile întreprinse sunt următoarele:
(1) aplicaţia trimite mesajul *REQUEST* (în zona de memorie partajată), mesaj care cuprinde o operaţie asupra obiectului OB1.
(2) managerul de tranzacţii TM_T1 recepţionează cererea şi, în urma analizei tabelelor sale, constatînd că obiectul nu a mai fost apelat în cadrul tranzacţiei T1, modifică tipul mesajului primit la FIRST_REQUEST şi îl trimite mai departe la CS_T1.
(3) CS_T1 transmite mesajul GET_OBJECT la serverul de nume, pentru a localiza obiectul.
(4) serverul de nume trimite răspunsul la CS_T1.
(5) CS_T1 începe implicarea obiectului de la distanţă în tranzacţie prin transmisia unui mesaj *NETWORK_BEGIN_TRANSACTION*.
(6) CS_T1 crează un proces fiu CS_T1_OB1 pentru a recepţiona mesaje de la nodul pe care se află obiectul.
(7) serverul de comunicaţii de pe nodul pe care se află obiectul, în urma recepţionării mesajului de la (5) crează un fiu CS_OB1_T1 care va gestiona comunicaţia cu obiectul OB1 în cadrul tranzacţiei de la distanţă T1 şi crează o zona de memorie partajată.
(8) CS_OB1_T1 (din nodul obiectului) transmite un mesaj *BEGIN_TRANSACTION* în canalul static al managerului obiectului cu informaţii despre zona de memorie partajată.
(9) managerul obiectului crează un fiu OB1_T1 care se va ocupa de tranzacţie; acesta se ataşează la memoria partajată şi transmite managerului obiectului OB1 un mesaj *TRANSACTION_ACCEPTED*.

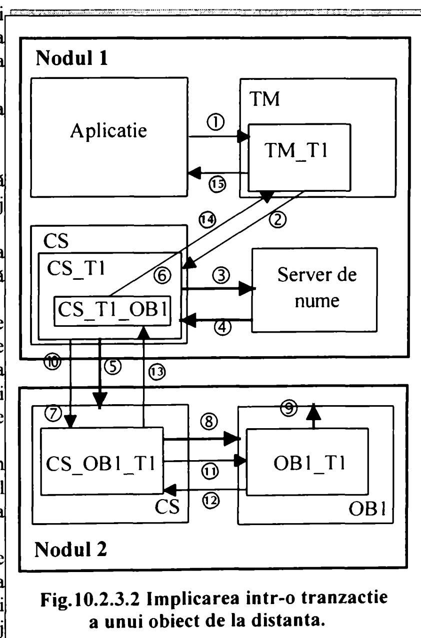


Fig.10.2.3.2 Implicarea intr-o tranzactie a unui obiect de la distanta.

10) CS_T1 transmite la CS_OB1_T1 cererea, într-un mesaj *FIRST_REQUEST*.
(11) CS_OB1_T1 transmite cererea la OB1_T1.
(12) OB1_T1 execută operaţia şi transmite mesajul *ACK_FIRST_REQUEST* către CS_OB1_T1.
(13) CS_OB1_T1 transmite răspunsul la CS_T1_OB1.
(14) CS_T1_OB1 transmite la TM_T1 răspunsul la cerere.
(15) TM_T1 transmite răspunsul la aplicaţie.

BUPT

10.2.4. Exemple de desfăşurare a protocolului two-phase commit

În cazul protocolului *two-phase commit* există trei aspecte care trebuie luate în considerare:
(1) acțiunile întreprinse de sistem în faza *prepare_to_commit*.
(2) acțiunile întreprinse de sistem în faza *commit*.
(3) acțiunile întreprinse de sistem în faza *abort*.

Pentru primul caz, acțiunile sunt reprezentate în fig. 10.2.4.1., în situația unei tranzacții care folosește două obiecte: OB1 local și OB2 la distanță.
Acțiunile sunt următoarele:

(1) aplicația transmite, în urma execuției funcției *EndTransmision()*, mesajul *END_TRANSACTION*.
(2) procesul TM_T1, care controlează tranzacția și recepționează mesajul *END_TRANSACTION*, crează un fiu TM_T1_P; acesta identifică obiectele participante la tranzacție: OB1 și OB2.
(3) TM_T1_P trimite către CS_T1 mesajul *PREPARE_TO_COMMIT* (pentru OB1).
(4) TM_T1_P trimite către CS_T1 mesajul *PREPARE_TO_COMMIT* (pentru OB2) și TM_T1_P se termină (exit()).
(5) CS_T1 transmite mesajul *PREPARE_TO_COMMIT* către OB1_T1.
(6) CS_T1 transmite mesajul *PREPARE_TO_COMMIT* către CS_OB2_T1.
(7) CS_OB2_T1 transmite la OB2_T1 mesajul primit, *PREPARE_TO_COMMIT*.
(8) OB2_T1 transmite *ACK_PREPARE_TO_COMMIT* la CS_OB2_T1.
(9) OB1_T1 transmite la CS_T1 mesajul *ACK_PREPARE_TO_COMMIT* (OB1).
(10) CS_T1 transmite la TM_T1 mesajul primit de la OB1_T1.
(11) CS_OB2_T1 transmite la CS_T1_OB2 mesajul primit de la OB2_T1.
(12) CS_T1_OB2 transmite la TM_T1 mesajul *ACK_PREPARE_TO_COMMIT* (OB2).

În momentul recepționării confirmarilor la mesajele *PREPARE_TO_COMMIT* de la toate obiectele implicate în tranzacție, managerul tranzacțiilor decide dacă tranzacția se poate temina prin *commit* sau trebuie terminată cu *abort*.

În cazul când tranzacția se poate termina cu *commit*, acțiunile și fluxul de mesaje din sistem sunt următoarele (fig. 10.2.4.2):

(1) TM_T1 crează un proces fiu TM_T1_C.
(2) TM_T1_C transmite mesajul *COMMIT* la CS_T1 (pentru obiectul OB1).
(3) TM_T1_C transmite mesajul *COMMIT* la CS_T1 (pentru obiectul OB2) și apoi se termină.
(4) CS_T1 transmite mesajul *COMMIT* la OB1_T1.
(5) CS_T1 transmite mesajul *COMMIT* la CS_OB2_T1.

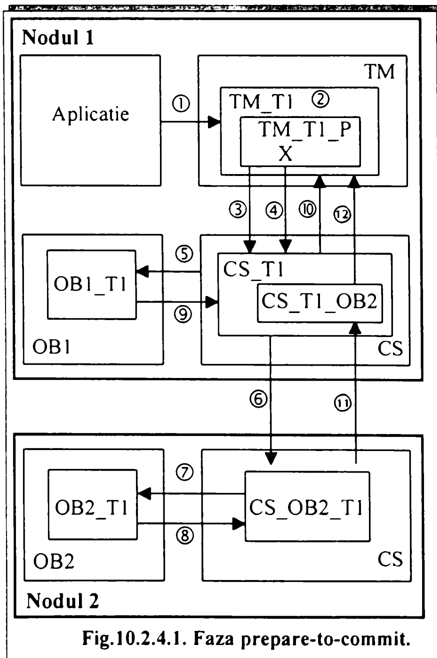


Fig.10.2.4.1. Faza prepare-to-commit.

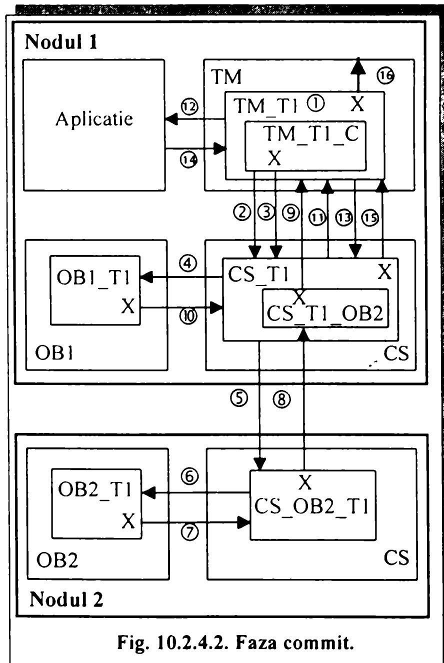


Fig. 10.2.4.2. Faza commit.

BUPT

(6) CS_OB2_T1 transmite mesajul *COMMIT* la OB2_T1.
(7) OB2_T1 tratează cererea *COMMIT*, transmite mesajul *ACK_COMMIT* la CS_OB2_T1 și apoi se termină.
(8) CS_OB2_T1 transmite mai departe mesajul *ACK_COMMIT* la CS_T1_OB2 și se termină.
(9) CS_T1_OB2 recepționează mesajul *ACK_COMMIT*, îl transmite mai departe la TM_T1 și se termină.
(10) OB1_T1 tratează cererea *COMMIT*, transmite mesajul *ACK_COMMIT* către CS_T1 și apoi se termină.
(11) CS_T1 transmite către TM_T1 mesajul recepționat, *ACK_COMMIT*.
(12) TM_T1 transmite la aplicație codul de eroare obținut în urma recepționării tuturor confirmărilor la *ACK_COMMIT*.
(13) TM_T1 transmite apoi la CS_T1 mesajul *TM_END*.
(14) după recepționarea răspunsului la *COMMIT*, aplicația trimite la TM_T1 mesajul *APL_END*.
(15) CS_T1, după recepționarea mesajului *TM_END* transmite la TM_T1 mesajul *CS_END*.
(16) TM_T1 transmite mesajul *TRANSACTION_ENDED* către TM, și apoi se termină.

Terminarea unei tranzacții prin abort generează în sistem o circulație de mesaje asemănătoare cu cea de mai sus (fig. 10.2.4.3) cu excepția faptului că se transmit mesaje *ABORT* (2, 3, 4, 5, 6, 7) și *ACK_ABORT* (8, 9, 10, 11, 12) în locul mesajelor *COMMIT* [i *ACK_COMMIT* și a faptului că răspunsul către aplicație se transmite imediat.

10.2.5. Exemple de tratare a situațiilor de recuperare

Există două situații de pierdere a contactului cu un nod implicat într-o tranzacție, care trebuie tratate în mod diferit: (1) cînd nodul cu care s-a pierdut contactul conține un obiect la distanță implicat într-o tranzacție; (2) când nodul cu care s-a pierdut contactul conține managerul de tranzacții coordonator al unei tranzacții. Există însă și o a treia situație în care trebuie raportată o eroare de tip "manager de obiecte inexistent", situație în care un nod care deține unul sau mai mulți manageri de obiecte nu mai răspunde la mesaje, dar care nu a fost implicat într-o tranzacție. În momentul în care un obiect de pe acest nod este implicat într-o tranzacție T1 în urma transmiterii mesajului *GET_OBJECT*, serverul de comunicații va primi ca răspuns un cod de eroare "manager de obiect inexistent". În acest caz se pozitionează în *TM_T1* un indicator *transErr* astfel încît să poată fi refuzate toate cererile de operații următoare din cadrul acelei tranzacții.

Acțiunile și fluxul de mesaje desfășurate de sistemul DOSTP într-o situație de tipul (2), când nodul pe care se afla coordonatorul unei tranzacții T1 nu mai răspunde la mesaje, și apoi este repus în funcțiune sunt următoarele (fig. 10.2.5.1):

(1) cînd nodul pe care se afla coordonatorul tranzacției nu mai este integrat în sistem, nodurile pe care se află obiectele implicate în tranzacții sesizează acest lucru prin faptul că, fie se primește un semnal de time-out în urma unei operații de recepție mesaj, fie se primește semnalul *SIGPIPE* (Unix) fie se returnează eroare la recepția sau emisia unui mesaj.
(2) CS_OB2_T1 transmite la OB2_T1 mesajul *INTERRUPT*.
(3) OB2_T1, după recepția mesajului *INTERRUPT*, transmite mesajul *ACK_INTERRUPT* și se termină.
(4) CS_OB2_T1, după recepția mesajului *ACK_INTERRUPT* distruge zona de memorie partajată și se termină.
(5) Se repune în funcțiune nodul coordonator

⇒

(A). Managerul de tranzacții T1, în urma analizei fișierului TMREC1, gasește tranzacția T1 neterminată și creează un proces fiu, TM_T1 care, în funcție de starea tranzacției, realizează următoarele:
(1) Dacă starea tranzacției se găsește a fi *ABORT* sau *WORKING*, atunci TM_T1 va forța execuția unei faze ABORT pentru toate obiectele implicate în tranzacție. Va crea un proces fiu care va trimite mesaje ABORT la toate obiectele și va aștepta răspunsuri de la acestea. Deci, mai întîi trimite mesajul ABORT (pentru obiectul 1) la CS.
(2) TM_T1 trimite mesajul *ABORT* (pentru obiectul OB2) la CS.
(3) CS, după recepția mesajului *ABORT* pentru obiectul OB1, creează un proces fiu CS_OB1.
(4) CS, după recepția mesajului ABORT pentru obiectul OB2, crează un proces fiu CS_OB2.
(5) CS_OB1 trimite un mesaj *GET_OBJECT* la serverul de nume, pentru a localiza obiectul.
(6) CS_OB2 trimite un mesaj *GET_OBJECT* la serverul de nume, pentru a localiza obiectul.
(7) Serverul de nume răspunde transmițînd localizarea OB1.
(8) Serverul de nume răspunde transmițînd localizarea OB2.
(9) CS_OB1 transmite mesajul *ABORT* la OB1.
(10) OB1 transmite la CS_OB1 confirmarea *ACK_ABORT*.
(11) CS_OB1 transmite la TM_T1 răspunsul *ACK_ABORT* și se termină.
(12) CS_OB2 transmite mesajul *ABORT* la serverul de comunicații CS de pe nodul obiectului OB2.
(13) CS primeste mesajul *ABORT* și creează un proces fiu CS_OB2_R.
(14) CS_OB2_R transmite mesajul la OB2 și așteaptă confirmarea.

BUPT

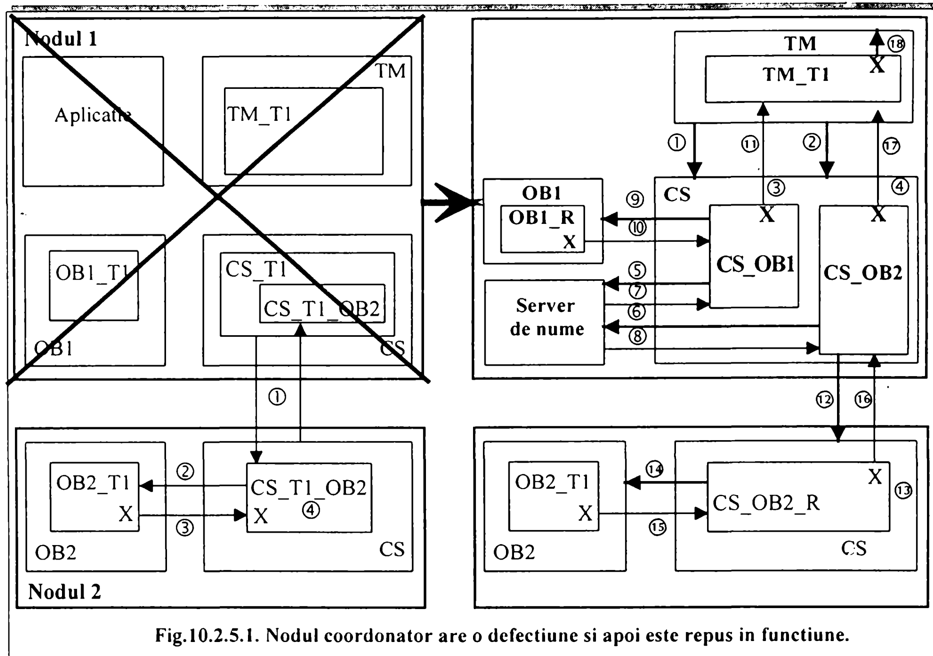


Fig.10.2.5.1. Nodul coordonator are o defectiune si apoi este repus in functiune.

(15) OB2 transmite ACK_ABORT la CS_OB2_R.
(16) CS_OB2_R transmite confirmarea la CS_OB2 și se termină.
(17) CS_OB2 transmite confirmarea la TM_T1 și se termină.
(18) TM_T1 primeste și confirmarea de la obiectul 2 și se termină.

B. Dacă starea tranzacției este *PREPARE_TO_COMMIT*, atunci, dacă în urma analizei fișierului TMREC2 se gasește starea tuturor obiectelor OK, se forteazâ faza *commit* iar circulația de mesaje este asemănătoare cu cea de mai sus 1-18, dar se lucrează cu mesaje *COMMIT* și confirmări pentru *COMMIT*. Dacă cel puțin un obiect este găsit în TMREC2 într-o stare diferită de OK atunci se forțează faza *abort* iar circulația mesajelor este aceeași ca mai sus 1-18.

C. Dacă starea tranzacției este *COMMIT* atunci se forțează faza *commit*.

Acțiunile și circulația mesajelor în DOSTP în cazul în care nodul pe care se află un obiect nu mai poate fi contactat sunt (fig. 10.2.5.2.):

(1) Cind nodul pe care se află un obiect nu mai este conectat în sistem (fie nodul are o defecțiune, fie linia de comunicație cu acel nod este defectă) nodul coordonator sesizează acest lucru fie obținînd erori la apelul unei operații de recepție sau emisie, fie primind un semnal SIGPIPE (Unix) fie un semnal de time-out.
(2) CS_T1_OB2 transmite un mesaj *INTERRUPT* la TM_T1 (în zona de memorie partajată) și se termină.
(3) TM_T1 analizează fișierul TMREC1 și în funcție de starea tranzacției decide execuția unei faze *commit* sau *abort* și crează un proces fiu TM_T1_R.
(4) TM_T1_R trimite un mesaj *INTERRUPT_ABORT* (sau *INTERRUPT _COMMIT*) la CS_T1 pentru obiectul OB1.
(5) TM_T1_R trimite un mesaj *INTERRUPT_ABORT (INTERRUPT_COMMIT)* la CS_T1 pentru obiectul OB2 și se termină.
(6) CS_T1 trimite un mesaj *ABORT* (sau *COMMIT*) la OB_T1.
(7) OB_T1 confirmă mesajul prin *ACK_ABORT* la CS_T1.
(8) CS_T1 transmite *ACK_ABORT* la TM_T1.
(9) TM_T1 distruge procesul care s-ar mai putea afla în execuție pentru a recepționa mesajele de la obiectul OB2 (CS_OB2_T1) și închide vechiul canal și crează un nou proces fiu CS_T1_R.
(10) CS_T1_R transmite un mesaj *GET_OBJECT* la serverul de nume pentru a determina localizarea obiectului OB2.
(11) Serverul de nume răspunde transmițînd localizarea lui OB2.
(12) CS_T1_R încearcă într-o buclă continuă să realizeze o conexiune cu serverul de comunicație.
(13) Se repune în funcțiune nodul care conține obiectul OB2.

BUPT

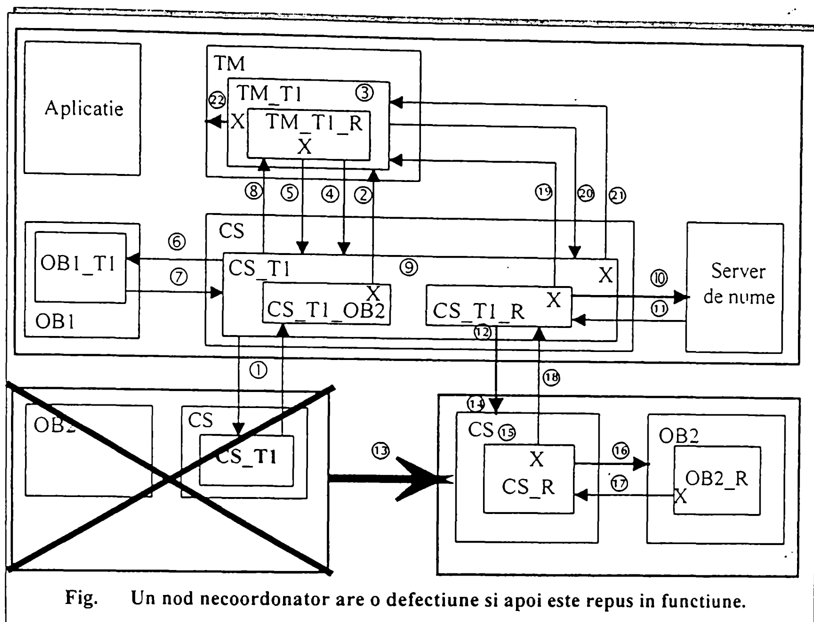


Fig. Un nod necoordonator are o defectiune si apoi este repus in functiune.

⇒

(14) Serverul de comunicaţii al nodului repus în funcţiune, CS, primeşte mesajul *ABORT* (sau *COMMIT*).
(15) CS crează un proces fiu CS_R care execută recuperarea.
(16) CS_R transmite obiectului OB2 mesajul *ABORT* (sau *COMMIT*).
(17) OB2 confirmă mesajul primit de la CS_R cu *ACK_ABORT* (sau *ACK_COMMIT*).
(18) CS_R transmite la TM_T1_R confirmarea primită şi se termină.
(19) CS_T1_R transmite confirmarea la TM_T1 şi se termină.
(20) TM_T1 transmite la CS_T1 mesajul *TM_END*.
(21) CS_T1 transmite la TM_T1 mesajul *CS_END* şi se termină.
(22) TM_T1 transmite managerului de tranzacţii *TRANSACTION_ENDED* şi se termină.

10.3. Exemplu de implementare al unui manager de obiecte

Subcapitolul următor prezintă modul de implementare, folosind programarea orientată pe obiecte, al unui manager de obiecte.

10.3.1. Descrierea managerului de obiecte

Scopul principal în realizarea managerilor de obiecte este ascunderea detaliilor de conectare cu sistemul tranzacţional. Managerii de obiecte sunt realizaţi astfel încît ei comunică numai cu serverul de comuniacţie local, fara a avea cunostinţă dacă serviciile le-au fost solicitate local sau de la distanţă.

Orice manager de obiecte care urmează să se integreze în sistemul DOSTP trebuie să implementeze următoarele cinci seturi de funcţii:

(1) *funcţia de conectare la sistem* care constă în transmiterea unui mesaj *RECORD* către managerul central, prin intermediul serverului de comunicaţii local
(2) *funcţia de eliminare din sistem* care constă în transmiterea unui mesaj *DELETE* către managerul central, prin intermediul serverului de comunicaţii local, în momentul cînd utilizatorul doreşte eliminarea obiectului din sistem
(3) *funcţia de lansare* a unei noi tranzacţii, la solicitarea unui client; această funcţie trebuie apelată la recepţionarea mesajului *BEGIN_TRANSACTION* în canalul static (soclu) al managerului de obiecte. Rolul său este de a genera un proces fiu (în Unix) sau un fir (în Windows) care să se ataşeze la zona de memorie partajată, să returneze confirmarea *ACK_FIRST_REQUEST* şi să deservească apoi cererile de operaţii asupra obiectului respectiv, recepţionate sub formă de pachete în zona de memorie partajată.
(4) *funcţia de recepţionare/ transmisie* a pachetelor din zona din/în zona de memorie partajată.
(5) *funcţiile necesare modului de lucru tranzacţional.* Acestea cuprind funcţiile care tratează mesajele *REQUEST, PREPARE_TO_COMMIT, COMMIT* şi *ABORT* primite ca pachete în zona de memorie

BUPT

partajată în timpul desfășurării normale a unei tranzacții și mesajul *INTERRUPT* primit ca urmare a pierderii legăturii cu nodul coordonator precum și funcțiile care tratează mesajele *COMMIT* și *ABORT* primite de la nodul coordonator în canalul fixat al managerului de obiecte în urma unei defecțiuni și apoi a repunerii în funcțiune a nodului (coordonator sau necoordonator).

10.3.1.1. Organizarea managerului de obiecte

Deoarece funcțiile din seturile 1-4 trebuie să le asigure orice manager de obiect, o primă consecință este că acestea, o dată scrise, realizarea unui nou manager de obiecte poate fi facută fără a se cunoaște detalii despre implementarea sistemului tranzacțional; trebuie implementate numai funcțiile din setul 5. A doua consecință este că un nou manager de obiecte poate fi testat separat, înainte de integrarea sa în sistemul tranzacțional. Soluția implementării obiectuale este utilă din ambele puncte de vedere.

Managerul de obiecte care s-a realizat pînă în momentul de față în sisteme Unix și Windows gestionează vectori cu elemente intregi.

După analiza seturilor de funcții de la 10.3.1. s-a optat pentru implementarea managerului de obiecte a următoarelor clase:

(1) o clasa de bază *CObject*, clasă abstractă, de la care, prin derivare să poată fi implementat orice tip de obiect. Această clasă trebuie să cuprindă funcțiile (virtuale) pe care un manager trebuie întotdeauna să le implementeze. Structura ei este următoarea:

```
class CObject :public CDebug, public CError {
public:
        CObjectManager(int debug=1, FILE *f =stdout); #constructor
        virtual int TreatRequest(PACKET *)=0;
        virtual int PrepareToCommit(void)=0;
        virtual int Commit(void)=0;
        virtual int Abort(void)=0;
        virtual int Interrupt(void)=0;
        virtual int Continue(int, int)=0;
        virtual int ObjectInit(void)=0;
        virtual int Finish(void)=0;
        virtual void SetTransNo(int, int)=0;
        virtual GetInitData(char *)=0;
        virtual void Startup(void)=0;
        virtual void Cleanup(void)=0;
};
class CError {
public:
        int m_errCode=0;
        virtual void Error(int)=0;
};
```

Pentru a ușura depanarea programelor, clasa *CObject* mai moșteneste încă două clase (vezi anexa D): clasa *CDebug*, al cărei rol este de a fixa fișierul în care se scriu mesajele pentru depanare (implicit stdout), de a determina identificatorul procesului curent (Unix) sau firului curent (Windows) care prefixează mesajele de depanare și de a fixa modul de lucru (cu sau fără depanare), și clasa *CError* care menține codul de eroare al ultimei operații și o metodă virtuală *Error()* care se particularizează la derivare.

Din definiția clasei *CObject* rezultă că orice manager de obiecte trebuie să implementeze funcțiile modului de lucru tranzacțional *(TreatRequest, PrepareToCommit, Commit, Abort)* și funcțiile care asigură recuperarea *(Interrupt și Continue)*. În plus, au mai fost introduse funcții pentru: execuția unor operații de inițializare la crearea obiectului *(ObjectInit)*, execuția unor operații de dealocare resurse la iesirea din sistem (*Finish*), execuția unor operații la lansarea/ terminarea unei tranzacții *(Cleanup, Startup)*, memorarea identificatorului și priorității unor tranzacții (*SetTransNo*) și preluarea unor date inițiale *(GetInitData)*.

(2) o clasă *CObjectManager* care să asigure integrarea unui obiect în sistemul tranzacțional, implementînd funcțiile din seturile 1, 2, 3 de la 10.3.1. Fiecarei instanțe de tip *CObject* i se asociază un obiect de tip *CObjectManager* care realizează legătura obiect-sistem tranzacțional.

(3) o clasă *CDispatcher* care asigură implementarea funcțiilor din setul 4 de la 10.3.1.

BUPT

10.3.1.2. Concurenţa la nivelul managerului de obiecte

În continuare, mecanismele de concurenţă şi de recuperare, care depind de tipul obiectelor implementate vor fi descrise pentru managerul de vectori implementat în DOSTP. Clasa care implementează comportarea acestui obiect, obţinută prin derivarea clasei *CObject* a fost denumită *CVector* şi este prezentată în anexa D. Asupra unui obiect de tip vector de elemente întregi pot fi realizate două operaţii: *Read* şi *Write*. Structura unei astfel de cereri şi a răspunsului la o cerere este următoarea:

```
typedef struct {
        int type;               /* tipul operatiei READ sau WRITE */
        int errorCode;          /* codul de eroare returnat in raspuns */
        int position;           /* pozitia in vector */
        int value;              / valoarea returnata la READ sau valoarea de scris la WRITE */
} VEC_REQUEST;
```

Vectorul este un *obiect persistent:* el este reprezentat în memoria externă printr-un fişier, al cărui nume este parametru al constructorului clasei. Pentru implementarea *listelor de intenţii* se folosesc două fişiere: un fişier care urmăreşte execuţia fazelor tranzacţiilor *(prepare-to-commit, commit sau abort)* şi un altul pentru memorarea operaţiilor executate de o tranzacţie şi care trebuie parmanentizate în faza *commit*. Numele acestor două fişiere se primesc tot ca parametri ai constructorului clasei *CVector*. În plus, pentru a evita încercările de recuperare a unor tranzacţii deja terminate, se mai lucrează cu un fişier care menţine identitatea tranzacţiilor încheiate.

La nivelul unui obiect, pot fi în lucru mai multe tranzacţii, deservite fiecare de cîte un proces separat (în Windows un fir separat); deoarece aceste procese trebuie să partajeze obiectul, soluţia adoptată este următoarea: programul principal prezintă o interfaţă grafică (XWindows sau Windows) care furnizează butoane pentru crearea unui obiect (vector), eliminarea unui obiect şi afişarea tranzacţiilor active pentru un obiect; în momentul creării unui obiect, în funcţia de tip *callback*, apelată în programul principal, se crează un obiect al clasei *CVector*. Constructorul acestui obiect mapează în memorie fişierul corespunzator obiectului persistent, deschide în modul *append* fişierele *log* şi alocă memorie pentru informaţiile de stare. Din punct de vedere al concurenţei, ultima operaţie prezintă importanţă. Informaţiile de stare ale obiectului vor fi partajate de toate procesele (sau firele) care deservesc tranzacţii pentru acel obiect. Ca atare, ele vor fi plasate într-o zona de memorie care trebuie să poată fi partajată de aceste proceses (sau fire) iar accesul la zonă să fie realizat în regim de excludere reciprocă, deci controlat prin semafoare.

Procesele care vor deservi tranzacţii pentru obiectul creat vor fi create de programul principal. În consecinţă, în Unix ele vor moşteni codul programului iniţial, deci vor cunoaşte adresa zonei de memorie partajată în care sunt depuse informaţiile de stare, la care trebuie însă *să se ataseze* pentru a avea acces. În Windows, lucrîndu-se cu fire şi mecanismul firelor nepunînd la dispoziţie conceptul de moştenire al codului ca în Unix, este necesară o copiere explicită a obiectului *CVector* din programul principal în firele care deservesc tranzacţii, pentru a se lucra cu o altă copie a obiectului, dar cu aceeaşi zonă de memorie pentru informaţiile de stare. De aceea, în anexa D, în definiţia clasei *CObject*, varianta Windows, apare o metodă *CopyObject* în plus faţă de varianta Unix. Structura şi semnificaţia informaţiilor de stare este următoarea:

```
struct STATE_INF {

TRANS_INF vTransId[MAX_TRANS_NO];  /* identificatorii si prioritatile tranzactiilor active */
short vBlockedTransId[MAX_TRANS_NO]; /*contine TRUE pentru tr. abortate prematur de o tr. mai prioritara */
short vReadPos[MAX_TRANS_NO][MAX_OBJ];     /* pentru fiecare tranzactie se retin pozitiile citite */
short vWritePos[MAX_TRANS_NO][MAX_OBJ];    /* pentru fiecare tranzactie se retin pozitiile scrise */
short vBlockedWPos[MAX_OBJ];               /* pozitia lock-urilor pentru scriere (write-lock-uri) */
short vBlockedRPos[MAX_OBJ];               /* pozitia lock-urilor pentru citire (read-lock-uri) */
OLD_TRANS vOldTrans[MAX_OLD_TRANS]; /* tr. in starea OK_PC, dar care nu au primit Commit sau Abort */
int VOldTransIdx;                        /* indice in vOldTrans */
};

typedef struct TRANS_INF {
        int transId;            /* identificatorul tranzac]iei */
        int priority;           /* prioritatea tranzac]iei */
} TRANS_INF;

typedef struct OLD_TRANS {
        int transNo;   /* identificatorul tranzac]iei */
        int priority;   /* prioritatea tranzac]iei */
        long inPC;    /*pozitia in fisierul PCLOG la care incep inregistrarile acestei tranzactii */
        short state;  /*starea tranzactiei:UNCHANGED,RECOVERED,RECOVERING,DONE,NOT_FOUND */
} OLD_TRANS;          /* starea tranzactiei este vazuta in raport de procesul de recuperare */
```


BUPT

Procesele (firele) care deservesc tranzacțiile sunt create de obiectul *CObjectManager* atașat unui obiect *CVector*; ele moștenesc (sau copiază) obiectul *CVector*, generînd astfel ***obiectele de lucru*** (care partajează informațiile de stare) și gestionează informații locale unei tranzacții precum starea curentă a tranzacției relativă la *prepare-to-commit* (*OK_PC, FAIL_PC, WORKING_PC*), operațiile Read și Write efectuate *(m_vRPos, m_vWPos)*, poziția în fișierul PCLOG a înregistrărilor corespunzătoare fazei *prepare-to-commit*.

Se va arăta in continuare modul cum au fost implementate funcțiile modului de lucru tranzacțional (declarate virtuale in clasa de baza) in clasa *CVector*, punîndu-se accentul pe aspectele de concurență de la nivelul obiectului:

(1) Funcția *TreatRequest* se apelează cînd se primește o cerere *Read* sau *Write* (pachet de tip *REQUEST*). O cerere de *Read* se tratează astfel: (a) se testează dacă cererea a survenit ***după*** primirea unui mesaj *PREPARE_TO_COMMIT* returnîndu-se in acest caz codul de eroare *EWRONG_STATUS*; (b) se testează dacă poziția din cerere nu depașește dimensiunea prestabilită pentru vector, returnîndu-se codul de eroare *EOVERFLOW*; (c) se testeaza daca tranzacția nu a fost blocată de o alta mai prioritară, returnîndu-se codul *ELOW_PRIORITY_ABORT* (se testează *m_pTrStatus -> vBlockedTransId[m_transIdx]*); (d) se testează dacă pe poziția solicitată nu există un *write-lock* (o altă tranzacție a ajuns în faza *prepare_to_commit* și va scrie această poziție) inspectîndu-se, pentru poziția solicitată, vectorul *m_pTrStatus -> vBlockedWPos* și returnîndu-se în acest caz *ELOCK_RECORD*; (e) dacă poziția nu a mai fost citită în această tranzacție, se memorează faptul ca s-a facut o citire, alocîndu-se o nouă intrare în vectorul local *m_vRPos[rIdx]* și în zona partajată; (f) se completeaza in pachetul de răspuns valoarea citită, luată fie direct din zona de memorie în care este mapat obiectul (daca tranzacția nu a scris încă aceasta poziție) fie din *m_vWPos[index].val.*

O cerere *Write* se tratează analog cu o cerere Read la punctele (a), (b), (c); la punctul (d) apare o diferență, în sensul ca daca poziția solicitată are un *read-lock*, operația de scriere nu eșuează deoarece scrierea efectivă se face la *commit* și dacă pînă atunci tranzacția care l-a citit se termină (șansele fiind mari, deoarece ea este în faza *prepare_to_commit*) totul este permis. Scrierea se face în copia locală a obiectului (*m_vWPos*) și se marchează și în zona partajata actualizarea acestei poziții.

În ambele operații, in caz de eșec, se marchează tranzacția în *m_pTrStatus -> vBlockedTransId[m_transIdx]* ca abortată și se deblochează articolele scrise/ citite de această tranzacție; totodată se face observația ca accesul la zona de memorie partajată se face în regim de excludere reciprocă controlat de semaforul *m_semObjId* (acelaşi pentru toate *obiectele de lucru*).

(2) Funcția *PrepareToCommit*() se apelează cînd se primește un pachet de tip *PREPARE_TO_COMMIT*. Ea realizează prima fază a protocolului *two-phase commit* astfel : (a) se testează dacă starea tranzacției este *WORKING_PC*; numai pentru această stare se execută *prepare_to_commit*, altfel se returnează *EWRONG_STATUS*; (b) dacă tranzacția a fost marcată ca abortată, se deblochează articolele scrise/citite de tranzacția respectiva și se returnează un cod de eroare *ELOW_PRIORITY_ABORTED*; (c) dacă cel puțin o poziție din cele citite în cadrul acestei tranzacții are un *write-lock*, sau invers, cel puțin o poziție scrisă are un *read-lock* se procedează ca la (b); (d) în acest moment, este sigur că tranzacția nu este în conflict cu alte tranzacții. Se verifică daca cel puțin o poziție pentru care tranzacția trebuie să obțină un *write-lock* nu a fost cumva citită de alta tranzacție *mai prioritara* (*nu* și invers, o poziție pentru care trebuie un *read-lock* dacă nu cumva a fost citită de alta tranzacție mai prioritară, deoarece această situație este permisă); în caz afirmativ, starea tranzacției devine *FAIL_PC* și se returnează *ETRY_AGAIN* (e) se obțin lock-urile (se incrementează pozițiile respective în *m_ptrStatus -> vBlockedWPos*, *m_pTrStatus -> vBlockedRPos* și se marchează ca blocate (abort) tranzacțiile care au citit pozițiile pe care tranzacția curentă le scrie; (f) se completează fișierul PCLOG și se completează deplasamentul său în *m_logPos*, se modifică starea tranzacției la *OK_PC* și se returnează *SUCCES*. Întreaga execuție a funcției *PrepareToCommit*() se execută sub controlul semaforului *m_semObjId* astfel incît actualizarea lock-ului și a fișierului se face în regim de excludere reciprocă.

(3) Funcția *Commit*() se apelează când se primește un pachet de tip COMMIT. Ea realizează faza *commit* a protocolului *two phase commit*, partea de server (vezi 7.3.1) astfel: (a) se testează dacă starea tranzacției este diferita de *OK_PC* returnîndu-se *EWRONG_STATUS* în acest caz; (b) se înscrie în fișierul ACLOG o înregistrare care să specifice identificatorul tranzacției, starea *COMMIT*, prioritatea ei, poziția listei de intenții în fișierul PCLOG; (c) se actualizează fișierul obiectului conform informațiilor obiectului de lucru local; (d) se înscrie in fișierul tranzacțiilor terminate identificatorul acestei tranzacții; (e) se marchează pentru ștergere înregistrările acestei tranzacții din PCLOG și ACLOG; (f) se eliberează *lock*-urile și intrarea corespunzatoare din *m_pTrStatus -> vTransId.*

(4) Funcția *Abort*() se apelează când se primește un pachet de tip *ABORT*; ea realizează faza *abort* a protocolului *two_phase commit*, partea de server (vezi 7.3.1) astfel: (a) dacă starea tranzacției este *OK_PC* se înscrie în fișierul ACLOG o înregistrare care să specifice identificatorul tranzacției, starea *ABORT*, prioritatea ei și poziția listei de intenții în fișierul PCLOG; (b) se eliberează *lock*-urile (dacă starea tranzacției era *PC_OK*), se demarchează articolele citite/scrise și se eliberează intrarea corespunzatoare din *m_pTrStatus -> vTransId.*

BUPT

10.3.1.2. Recuperarea obiectelor

La nivelul obiectelor, există două aspecte ale recuperării obiectelor (vezi și 10.2.5):

(1) primul aspect apare cînd se lansează în execuție un manager al unui obiect, după ce nodul obiectului pierduse contactul cu nodul coordonator al unei tranzacții în care era implicat obiectul. În această situație managerul de obiecte analizează fișierele de log și ia fie decizia de *abort*, fie de *commit*, fie de așteptare a deciziei coordonatorului;

(2) Al doilea aspect apare cînd managerul de obiect primește un pachet *INTERRUPT* de la serverul de comunicație ca urmare a pierderii legăturii cu coordonatorul unei tranzacții.

Primul aspect este tratat de funcția *ObjectInit*() a clasei *CObject* și deci și a managerului *CVector*. Funcția *ObjectInit* se apelează imediat după crearea unui obiect și are trei scopuri:

(1) termină tranzacțiile care au recepționat de la coordonator comanda *Abort* sau *Commit*, dar, datorită unei defecțiuni, nu au putut-o realiza;

(2) pastrează în fișierele de log numai tranzacțiile cu înregistrari valide, ștergînd tranzacțiile pentru care s-a realizat faza *Commit* sau *Abort*;

(3) înregistrează în lista *m_pTrStatus -> vOldTrans* informații relative la tranzacțiile care nu au putut fi terminate (sunt în starea *OK_PC*, dar nu s-a înregistrat în ACLOG comanda dată de coordonatorul tranzacției) și face blocările de obiecte necesare; pentru fiecare articol scris de o tranzacție, obține un *write_lock* și pentru fiecare articol citit obține un *read_lock*.

Al doilea aspect este tratat de funcția *Interrupt*() și *Continue*() ale clasei *CObject* (și deci și ale managerului *CVector*). Funcția *Interrupt*() este apelată atunci când serverul de comunicații detectează întreruperea conexiunii cu coordonatorul tranzacției (și aplicația se află la distanță). Ea realizează următoarele operații în regim de excludere reciprocă:

(1) demarchează articolele scrise/citite de această tranzacție;

(2) dacă tranzacția este în starea *OK_PC*, atunci se înscrie această tranzacție în vectorul tranzacțiilor neterminate (*m_pTrStatus -> vOldTrans*);

(3) eliberează intrarea alocată acestei tranzacții în *m_pTrStatus -> vTransId*.

Funcția *Continue*() se apelează cînd, în urma repunerii în funcțiune a nodului coordonator se primește comanda *Abort* sau *Commit* în canalul static. În acest caz, acțiunile care trebuie realizate depind de starea tranzacției neterminate și sunt următoarele:

(1) dacă tranzacția este găsită în fișierul tranzacțiilor încheiate, *m_tf*, atunci ea a fost terminată și se transmite confirmarea fără nici o altă acțiune;

(2) dacă tranzacția este gasită în tabelul *m_pTrStatus -> vTransId* atunci tranzacția este încă activă; această situație poate apare cînd procesul care tratează tranzacția nu a tratat încă pachetul *INTERRUPT*, dar coordonatorul a fost repus foarte repede în funcțiune. În acest caz, se așteaptă pînă tranzacția nu mai apare ca activă, dar este, în schimb, înscrisă în *m_pTrStatus -> vOldTrans* și se iau următoarele măsuri: (2.1) dacă starea tranzacției memorată în *m_pTrStatus -> vOldTrans* este *RECOVERING*, atunci rezultă că mai este în lucru un proces (sau fir) care are în lucru recuperarea tranzacției respective și, ca atare, procesul curent se va termina, lasînd ca răspunsul să fie dat de procesul în lucru; (2.2) dacă starea tranzacției este *UNCHANGED*, atunci procesul curent recuperează tranzacția; (2.3) dacă starea tranzacției este *DONE* înseamnă că deja un proces a executat recuperarea și se transmite confirmarea; (2.4) dacă tranzacția nu este gasită în nici o stare din cele de mai sus, înseamnă că tranzacția a fost abortată și se transmite confirmarea.

Pentru recuperarea propriu-zisă a unei tranzacții în starea *OK_PC* a fost construită o clasă specială *CRecovery* care furnizează următoarele funcții:

(1) *ResumeCommit*() pentru actualizarea versiunii persistente a obiectului conform actualizărilor memorate în fișierul log PCLOG și actualizarea apoi a fișierelor log;

(2) *ResumeAbort*() pentru actualizarea fișierelor de log;

(3) *Commit*() pentru execuția tuturor acțiunilor funcției *Commit* din *CVector*, dar, pe baza informațiilor de actualizare găsite în fișierul PCLOG;

(4) *Abort*() pentru execuția tuturor acțiunilor funcției *Abort* din *CVector*.

Funcțiile (1) și (2) se apelează cînd trebuie continuată comiterea/abortarea unei tranzacții (care nu a putut fi terminată din cauza unei defecțiuni, primul aspect al recuperarii), în funcția *ObjectInit*()), după crearea unui obiect *Crecovery* caruia i se transmit ca parametri (constructorului sau): fișierele de log (numele și pointeri la structura lor de tipul FILE), pointerul la structura de stare a obiectului (*m_pTrStatus*) deplasamentul articolelor corespunzătoare tranzacției în fișierul PCLOG, prioritatea și identificatorul tranzacției, semaforul pentru controlul accesului exclusiv.

Funcțiile (3) și (4) se apelează când se primește comanda *Abort* sau *Commit* în canalul static al managerului de obiecte (al doilea aspect al recuperarii, în funcția *Continue*()).

BUPT

10.3.2. Conectarea obiectelor în sistemul DOSTP

S-a aratat la 10.3.1.1 ca, pentru conectarea obiectelor în sistem, au fost prevazute două clase: *CObjectManager* și *CDispatcher*. În acest subcapitol vor fi prezentate aceste două clase și modul de integrare automată a obiectelor în sistemul tranzacțional.

10.3.2.1. Descrierea clasei CObjectManager

Clasa *CObjectManager* (a cărei definiție este prezentată în anexa D) a fost introdusă în vederea următoarelor două scopuri: (1) să realizeze comunicarea prin canalul static dintre un manager de obiecte cu serverul de comunicații (canalul static este implementat sub forma de soclu *Unix* pentru versiunea Unix și soclu *Inet* pentru versiunea Windows); (2) să pună la dispoziție funcții de vizualizare a tranzacțiilor active. În vederea atingerii acestor scopuri au fost introduși membri precum porturile și soclurile pentru comunicare, pointer la instanța *CObject* asociată, o listă de tranzacții care folosesc instanța *CObject* și alți membri necesari și specifici implementării Unix (widget label, quit, box) sau Windows (semafoare folosite la copierea informațiilor transmise firelor create la inițierea unei noi tranzacții).

În ceea ce privește funcțiile puse la dispoziție de această clasă se enumeră: (1) funcția *ConnectOjectToSystem*() care realizează transmisia mesajului *RECORD* la managerul central (via server de comunicații) și punerea soclului în asteptare de recepție mesaje; (2) *DisconnectObjectFromSystem*() care realizează transmisia mesajului *DELETE* la managerul central (via serverul de comunicații), inchiderea soclurilor de comunicare, apelul *Finish*() al instanței obiectului asociat *CObject* pentru dealocarea resurselor (eliberarea semafoarelor pentru accesul exclusiv la informațiile de stare și pentru accesul exclusiv la vectorul tranzacțiilor neterminate *vOldTrans*); (3) funcția *GetNextRequest*() care realizează: (3.1) acceptarea unei noi conexiuni și preluarea primei cereri din soclu (în mod normal de tip *BEGIN_TRANSACTION* sau de tip *COMMIT/ABORT*); (3.2) dacă cererea este de tip *BEGIN_TRANSACTION*, se crează un proces (sau fir) care declară un obiect de tip *CDispatcher* care va realiza toată conversația cu serverul de comunicații prin zona de memorie partajată; (3.3) dacă cererea este de tip *ABORT/COMMIT* (trimisă de un manager de tranzacții care vrea să recupereze o tranzacție) se crează un proces (sau fir) care declară un obiect de tip *CDispatcher* și se apelează metoda *Continue* a acestuia; aceasta metodă, după câteva operații de rutină, va apela în cele din urmă metoda *Continue* a instanței *CObject* asociate.

10.3.2.2. Descrierea clasei CDispatcher

Clasa *CDispatcher* (a cărei definiție este prezentată în anexa D) a fost introdusă pentru a implementa funcțiile de preluare/transmitere a pachetelor din zona de memorie partajată. Se crează cîte o instanță a acestei clase la fiecare inițiere a unei tranzacții (la primirea unui mesaj *BEGIN_TRANSACTION*) sau la fiecare cerere *ABORT/COMMIT* pentru o tranzacție nedeterminată. Instanțele se crează în cadrul unui proces (sau fir) separat care deservește desfășurarea normală a tranzacției respectiv recuperarea tranzacției.

În cadrul acestei clase sunt prevăzute funcții pentru: (1) atașarea și detașarea la zona de memorie partajată prin care are dialogul cu serverul de comunicații; (2) blocare pînă la recepționarea unui pachet din zona de memorie partajată; (3) preluarea unui pachet din zona de memorie partajată, apelul funcției de tratare a cererii din instanța *CObject* atașată și returnarea răspunsului la cerere tot în zona de memorie partajată; (4) recuperarea unei tranzacții folosind funcția *Continue* din instanța *CObject* asociată. Spre deosebire de primele 3 clase de funcții, pentru recuperarea unei tranzacții, dialogul cu serverul de comunicații se face printr-un canal static (soclu *Unix* sau *Inet*) și nu printr-un canal creat în mod dinamic (zona de memorie partajată).

10.3.2.3. Integrarea automată a obiectelor în sistemul tranzacțional

Pentru a ușura scrierea codului pentru integrarea obiectelor în sistemul tranzacțional s-au generat patru macrodefiniții. Acestea sunt definite în anexa D și au următoarele scopuri:

(1) Macrodefiniția *INCLUDE_SYSTEM*, inclusă in secțiunea "public" a definiției unei clase (ea este inclusă de fapt in definiția clasei CVector : public INCLUDE_SYSTEM (CVector)) are efectul de a introduce automat în definiția clasei a inca citorva membri. Pentru varianta Unix acești membri sunt: un pointer la instanța *CObjectManager* asociată obiectului (creată dinamic), o funcție *friend init* care primește ca parametrii contextul aplicației XWindows obținut în programul principal și identificatorul widget-ului in care instanța *CObjectManager* atașată obiectului va face afișările. Pentru varianta Windows acești membri sunt: portul pe care managerul așteaptă cereri, semaforul folosit la crearea firelor în care acționează instanța *CDispatcher*, și două funcții *friend*: *Init*() și *CreateThreads*().

(2) Macrodefiniția *DEFINE_SYSTEM* asigură definirea funcțiilor prietene de mai sus.

(3) Macrodefiniția *START_SYSTEM* asigură conectarea unui obiect la sistemul tranzacțional. Ea constă numai în apelul funcției *Init*() definită prin *DEFINE_SYSTEM*.

BUPT

(4) Macrodefiniția *STOP_SYSTEM* asigură deconectarea unui obiect de la sistemul tranzacțional. Ea constă numai în apelul metodei *DisconnectObjectFromSystem*() a instanței *CObjectManager*. Folosind aceste definiții, un manager de obiecte (programul principal) poate fi scris astfel:

```
void Quit(Widget w,XtPointer client_data,XtPointer call_data);
void Add(Widget w,XtPointer client_data,XtPointer call_data);

CVector v1 ("v1","data1","PCLOG1","ACLOG1","Trans","Status"),*v2;  // se creaza un obiect static

XtAppContext app_cont;
Widget shell ,pane,box,quit,add;
FILE *f1,*f2;

void main (int argc,char** argv)
{
 shell = XtVaAppInitialize( &app_cont, "XBox", NULL, 0, &argc,argv, NULL,NULL );
 pane = XtVaCreateManagedWidget("pane",panedWidgetClass, shell,  NULL);
 box = XtVaCreateManagedWidget("box", boxWidgetClass,pane,NULL);
 quit = XtVaCreateManagedWidget("quit",commandWidgetClass,box,NULL);
 add = XtVaCreateManagedWidget("addv2",commandWidgetClass,box,NULL);

 f1=fopen("debug1","w+");          // deschiderea unui fisier pentru depanare
 setbuffer(f1,0,0);                // mod de lucru fara buferare
 v1.Init(1,f1);                    // stabilirea lui f1 ca fisier in care se vor inscrie mesajele de depanare
 v1.ObjectInit();                  // faza de recuperare initiala pentru tranzactiile care pot fi terminate

 START_SISTEM (&v1,app_cont,pane);                 // conectarea la sistem

 XtAddCallback(quit,XtNcallback,Quit,NULL);        // functie apelata la stergerea  unui obiect
 XtAddCallback(add,XtNcallback,Add,NULL);          // functie apelata la adaugarea unui obiect de acelasi tip
 XtRealizeWidget(shell);
 XtAppMainLoop(app_cont);
}

void Add(Widget w,XtPointer client_data,XtPointer call_data)
{
 v2=new CVector("v2","data2","PCLOG2","ACLOG2","Trans","Status");

 f2=fopen("debug2","w+");          // deschiderea unui fisier pentru depanare
 setbuffer(f2,0,0);                // mod de lucru fara buferare
 v2->Init(1,f2);                   // stabilirea lui f1 ca fisier in care se vor inscrie mesajele de depanare
 v2->ObjectInit();                 // faza de recuperare initiala pentru tranzactiile care pot fi terminate

 START_SISTEM(v2,app_cont,pane);                   // conectarea la sistem

 XtRemoveCallback(add,XtNcallback,Add,NULL);
 XtDestroyWidget(add);
}

void Quit(Widget w,XtPointer client_data,XtPointer call_data)
{
 if (v2) STOP_SISTEM(v2);          // deconectarea de la sistem a obiectului v1
 v2=&v1;

 STOP_SISTEM(v2);                  // deconectarea de la sistem a obiectului v2
 exit(0);
}
```


BUPT

10.3.3. Particularități ale implementării în sistemele Unix și Windows

Diferențele între implementarea unui manager de obiecte în sistemul Unix și în sistemul Windows provin în principal din faptul că în sistemul Unix se lucrează cu procese, care moștenesc codul procesului părinte, și care, după creare, nu mai au acces la acesta, iar în Windows se lucrează cu fire. Cele trei clase de bază prezentate la 10.3.2 sunt aproape identice în ambele implementari. Diferă însă lucrul cu semafoare și cu zonele de memorie partajată. Se enumeră în continuare cîteva diferențe între cele două implementări:

(1) mesajele *BEGIN_TRANSACTION* conțin numai pentru versiunea Unix identificatori pentru semafoare și zona de memorie partajată la care instanța *CDispatcher* trebuie să se atașeze.

(2) comunicația între serverul de comunicații și managerul de obiecte, la nivelul mesajelor *BEGIN_TRANSACTION, ABORT, COMMIT* se face printr-un soclu *Inet* în versiunea Windows și printr-un soclu *Unix* în versiunea Unix.

(3) instanțierile clasei *CDispatcher* se fac în procese separate în versiunea Unix și în fire separate în versiunea Windows.

(4) În versiunea Windows firele nu moștenesc codul părintelui și de aceea, în clasa *CObject* trebuie introdusă o metodă *CopyObject* care să asigure copierea instanței *CObject* de pe firul principal în firele similare, create la inițierea unei tranzacții.

(5) În versiunea Unix, deoarece se crează procese care moștenesc descriptorii de fișiere ai procesului părinte, aceștia trebuie închiși în procesele fii.

(6) dimensiunea articolelor din fișierelor text PCLOG și ACLOG este mai mică cu un octet în fișierele Unix.

(7) În Unix se lucrează cu funcția *fsync*() pentru siguranța scrierii pe disc, iar în Windows cu *commit*().

(8) macrodefinițiile pentru integrarea automată în sistem a obiectelor diferă.

(9) fișierele folosite sunt setate fara buferare (pentru ca operațiile *fprintf* de scriere să se facă direct pe fișiere) cu funcția *setbuf*() în Windows și cu funcția *setbuffer*() în Unix.

(10) maparea fișierului obiectului (versiunea persistentă) se face în Unix cu apelul:

```
m_pData = (int *)mmap(0 ,size , PROT_READIPROT_WRITE ,MAP_FILE I MAP_SHARED ,m_df,0);
        if ( (int)m_pData==-1) Panic("ObjMan:  mmap object error")
```

iar în Windows cu apelurile:

```
m_df=CreateFile(m_dataFile,GENERIC_READIGENERIC_WRITE,FILE_SHARE_READI FILE_SHARE_WRITE,
                                        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (m_df == INVALID_HANDLE_VALUE)
        Panic("ObjMan: CreatFile object error")
if((m_hData=CreateFileMapping(m_df, NULL,PAGE_READWRITE,0,sizeof(int)*MAX_OB,m_dataFile))==NULL)
        Panic("ObjMan: MappingFile object error")
if ( (m_pData = ( int*)MapViewOfFile (m_hData, FILE_MAP_WRITE, 0, 0,sizeof(int)*MAX_OB ) ) == NULL )
        Panic("ObjMan: MapViewOfFile object error")
```

(11) În programul principal din versiunea Unix trebuie captat semnalul SIGCHLD, pentru a se executa apelul *wait*() în momentul în care se termină un proces fiu (se evită apariția proceselor *zombie*).

BUPT

10.3.4. Exemplu de aplicaţie care apelează managerii de obiete de tipul proiectat

Orice aplicaţie care foloseşte managerii de obiecte de tipul proiectat trebuie să folosească modelul prezentat la 9.4.2. Acesta înseamnă că pentru lucrul cu sistemul tranzacţional secvenţa de instrucţiuni care trebuie executată trebuie să cuprindă:

```
            /* conectarea la sistemul tranzac]ional DOSTP */
(1) ...
            /* initierea unei tranzac]ii */
(2) ...
            /* apeluri de operatii sincrone sau asincrone asupra unor obiecte locale sau la distanta */
            /* receptionarea rezulattelor */
(3) ...
            /* terminarea operatiilor asincrone lansate */
(4) ...
            /* încheierea tranzac]iei cu commit sau abort
(5) ...
            /* deconectarea de la sistem */
(6) ...
```

Etapele 1-4 pot fi realizate, într-o aplicaţie foarte simplă astfel:

```
(1) if (errorCode = InitDOSTP ("Test") {              /* încercare de conectare la sistem */
            printf ("Init error = %d\n", errorCode);    /* încercare nereusita */
            close DOSTP();                              /* în acest caz, deconectare de lasistem */
    }

(2) if (errorCode = BeginTransaction () {                    /* încercare de initiere a unei tranzac]ii */
            printf ("BeginTransaction errorr = %d\n", errorCode); /* încercare nereusita */
            close DOSTP();                                   /* în acest caz, deconectare de la sistem */
    }

(3)     VEC_REQUEST vrequest;/* cerere de executie a unei operatii */
        int status;
        vrequest.position = position;
        vrequest.value = value;
        errorCode = AsyncRequest ("v1", &vrequest, & request, &status, TIME_OUT)
        if (errorCode)
                printf ("AsyncRequest error = %d\n", errorCode);
        ..................................................................
        errorCode =SyncRequest ("v2, &vrequest, &vrequest, TIME_OUT)
        if (errorCode)
                printf ("SyncRequest error = %d\n", errorCode);


(4)     errorCode = Syncronize();
        if (errorCode)
                printf ("Syncronize error = %d\n", errorCode);


(5)     again:
        errorCode = EndTransmision();              /* se încerca terminarea tranzac]ieie */
        if (errorCode = ETRY_AGAIN) {
                sleep(1);                          /* se asteapta 1 secunda */
                if (retry++ < MAX_RETRY)
                        goto again;                /* se reîncearca commit */
                else
                        CloseDOSTP();  /* daca nu se reuseste dupa MAX_RETRY încercari, deconectare*/
                                       /* de la sistem */
        }

(6)     CloseDOSTP():
```


BUPT

10.4. Concluzii și direcții de dezvoltare ulterioară

În urma analizei proiectării și implementării sistemului DOSTP pot fi enunțate următoarele concluzii:

(1) Sistemul DOSTP constituie un suport software pentru dezvoltarea de aplicații distribuite. Arhitectura sa nu este complet distribuită deoarece se prevede un manager central pe un nod fixat în sistem. Introducerea managerului central este un compromis acceptat din motive de performanță; dacă nu s-ar fi folosit managerul central, atunci, pentru obținerea, într-o manieră complet distribuită, a priorității și identificatorilor tranzacțiilor, unici în intreg sistemul, ar fi trebuit folosită o metodă prezentată la 4.3.2, 4.3.3 sau 4.3.4, ceea ce implică însă o circulație mare de mesaje.

(2) Sistemul a fost proiectat astfel încît să ofere o eficiență maximă. Comunicațiile locale între componentele sistemului, în timpul desfășurării unei tranzacții, cînd au loc multe transferuri de date se fac prin intermediul unei zone de memorie partajate. Metoda oferă rapiditate dar are dezavantajul că necesită mai multe resurse auxiliare (semafoare pentru controlul accesului în regim de excludere reciprocă, canale pentru stabilirea legăturii) și un protocol relativ lent de conectare (se creează mai întîi zona de memorie partajată, semafoarele pentru controlul accesului, apoi se transmit identificatorii acestora prin canale FIFO). În schimb, după realizarea conectarii, transferul de date decurge foarte rapid, compensîndu-se pierderea de timp de la început.

(3) La implementarea versiunii Unix și a versiunii Windows au trebuit să fie rezolvate diferențele între funcțiile ce lucrează cu soclurile, cu memoria partajată și cu semafoarele. În Unix se lucrează cu seturi de semafoare, fiecare set fiind caracterizat de un identificator propriu; dacă un proces posedă acest identificator, atunci el poate folosi setul de semafoare. În Windows nu există seturi de semafoare; semafoarele identificîndu-se prin nume; dacă un proces cunoaște numele respectiv, atunci el poate folosi semaforul respectiv. Analog stau lucrurile și cu zonele de memorie partajată: în Windows, zonele de memorie partajată (eventual cu fișiere mapate în memorie) se identifică prin nume iar în Unix printr-un identificator.

(4) Diferentele majore între versiunile Unix și Windows se datorează facilităților diferite în lucrul în procese și fire oferite de Unix respectiv Windows; acestea nu au afectat logica programelor dar au introdus necesitatea unor operații de copiere și sincronizare în plus.

(5) În Windows nu se pot folosi socluri pentru comunicarea locală (socluri Unix), de aceea, în versiunea Windows, procesele de pe acelasi nod comunică prin socluri TCP. Un soclu Unix este caracterizat printr-un nume de "port", care respectă regulile de nume ale fișierelor, spre deosebire de un soclu de tip Inet (TCP) care este caracterizat printr-un nume de port. În consecință, obiectelor de pe Windows li se asociază pentru a fi contactate un șir de caractere drept port iar celor de pe Unix un număr.

(6) Pe parcursul realizării testelor, au fost sesizate aspecte nespecificate ale apelurilor sistem: dacă, de exemplu, un server de comunicație de pe un nod Unix execută apelul *write()* și serverul de comunicație pereche Windows are o defecțiune sau este terminat forțat, atunci serverul de comunicație primește semnalul SIGPIPE, sesizînd apariția defecțiunii; dacă nodul care are defecțiunea este nod Unix atunci apelul *write() reuseste*, cu toate că partenerul de comunicație a dispărut (dispariția va fi sesizată la următorul apel *read()*). Apelul funcției sistem *read()*, în aceleași condiții sesizează dispariția nodului pereche, returnînd valoarea -1.

(7) Pentru situațiile de recuperare nu se mai folosește memoria partajată, avînd în vedere că se transmit doar cîteva comenzi și confirmări (traficul de mesaje este mic).

(8) Pe sisteme uniprocesor diferențele între timpii de execuție ale aplicațiilor care efectuează cereri asincrone către obiecte locale și între cei ai aplicațiilor care efectuează aceleași cereri sincron sunt mici. Avantajul prelucrării asincrone apare în cazurile în care aplicațiile execută cereri asincrone asupra mai multor obiecte de la distanță (prelucrările se fac paralel).

(9) Prin maniera de conectare automată a managerilor de obiecte la sistem (folosind clasele definite la 10.3.2.1 și 10.3.2.2 și macrodefinițiile de la 10.3.2.3) managerii de obiecte pot fi implementați și testați separat de sistemul DOSTP.

(10) Pentru a pune la dispoziția componentelor sistemului anumite informații precum localizarea managerului central, directoare pentru fișiere temporare se folosește un *fișier de configurare*. Folosind acest fișier, pot fi făcute modificări fără a fi necesară recompilarea surselor.

(11) O posibilă extindere a sistemului DOSTP ar consta în introducerea drepturilor de acces la sistemul DOSTP. Acest lucru poate fi realizat specificîndu-se o parolă în funcția de conectare la sistem *InitDOSTP()*.

(12) În varianta actuală a sistemului DOSTP se presupune că nodul pe care se afla managerul central nu se defectează. O extindere a sistemului ar putea fi facută și în acest sens; astfel, se pot folosi fișiere log la nivelul managerului central care conțin informații despre nodurile conectate la sistem și obiectele introduse în sistem, și care se pot citi la relansarea managerului central, și eventual adăugat un protocol de interogare al acestora, dacă mai sunt în funcțiune.

BUPT

Capitolul 11
Concluzii

11.1 Concluzii generale

Scopul acestei teze, intitulată "Contribuţii la proiectarea sistemelor de operare distribuite" este de a-şi aduce contribuţia, într-o manieră originală, la tratarea unor probleme şi aspecte ale proiectării sistemelor de operare distribuite, un domeniu relativ nou pentru informatica din ţara noastră. Autorul şi-a propus ca în această lucrare să abordeze atît aspecte teoretice ale acestui domeniu, considerînd că domeniul respectiv se află încă în faza de identificare şi încercare de soluţionare a problemelor sale fundamentale, dar şi aspecte practice constînd în implementarea unor API-uri destinate să asiste utilizatorii în dezvoltarea de aplicaţii în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare.

Lucrarea este împărţită în două mari părţi: prima parte, care cuprinde primele cinci capitole, are ca obiective principale prezentarea problemelor specifice unui sistem de operare distribuit, accentuîndu-se problemele de comunicare şi sincronizare, paradigmele interacţiunii dintre procese precum şi legătura sistem de operare distribuit-aplicaţie distribuită. În cadrul primei părţi se prezintă modul de proiectare şi implementare a două susbsisteme software. Scopul acestor subsisteme software este: (1) controlul accesului concurent la o bază de date în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare; (2) furnizarea unor API-uri pentru comunicaţii de grup în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare.

A doua parte a tezei, care cuprinde capitolele 6-10, reprezintă un studiu sinteză asupra modalităţilor şi criteriilor de proiectare a sistemelor de programare distribuită bazată pe obiecte (SPDBO) precum şi proiectarea şi implementarea unui suport software bazat pe modelul tranzacţiilor atomice pentru SPDBO în reţele Unix şi Microsoft Windows (Win32).

În urma parcurgerii celor 10 capitole ale lucrării, pot fi enumerate următoarele concluzii:

Partea I: SISTEME DISTRIBUITE

Capitolul 1: Sisteme de operare distribuite. Suportul oferit prelucrarilor distribuite: probleme specifice, modele de proiectare.

Sistemul de operare asigură un mediu de execuţie pentru programele aplicative, asumîndu-şi în contul acestora toate sarcinile necesare, începînd de la activităţi de gestiune, partajare şi control al accesului la diverse resurse, pînă la asigurarea unor interfeţe uniforme şi a unor niveluri diferenţiate de abstractizare/virtalizare.

În capitolul 1 se abordează şi problema interacţiunii prelucrare distribuită-sistem de operare, prezentîndu-se, prin sistemul TDL, o modalitate de conectare a programelor în reţele Unix şi Windows. În mod curent, utilizatorii unui mediu eterogen dezvoltă programe în manieră tradiţională: se scriu procese separate ca programe în diferite limbaje (Pascal,C) implementate pe anumite procesoare şi apoi printr-un program separat se încarcă aceste programe pentru execuţie. Deseori, aceste programe sunt scrise astfel încît înglobează în ele specificaţii privind dispunerea lor pe procesoare sau chiar privind structura reţelei. Acest mod de lucru face dificilă sau chiar imposibilă utilizarea aplicaţiei într-un mediu diferit, complică o dezvoltare ulterioară a aplicaţiei care ar schimba structura programelor şi suportul pentru siguranţă la defecte (prin restartarea unei noi copii a programului pe un procesor diferit). O manieră nouă de abordare a acestor probleme este de a separa activitatea de programare a programelor de activitatea de cooperare a lor în codul aplicaţiei.Vor exista astfel două nivele de programare în cadrul unei aplicaţii: *un nivel de programare al programelor componente ale unei aplicaţii şi un nivel de programare a aplicaţiei care utilizează programele dezvoltate anterior.* O aplicaţie distribuită în cadrul sistemului proiectat, TDL, constă într-un set de taskuri care comunică prin cozi de mesaje. . Descrierile taskurilor şi declaraţiile de tipuri specifică modul în care taskurile sunt lansate în execuţie şi executate ca procese concurente, tipul datelor prin care comunică procesele, precum şi cozile de date necesare proceselor producătoare şi consumatoare. Există trei faze distincte în dezvoltarea unei aplicaţii distribuite utilizînd limbajul TDL: crearea unei biblioteci de taskuri, crearea unei aplicaţii utilizînd biblioteca de taskuri şi execuţia aplicaţiei. În mediul de execuţie există trei componente active: *taskurile de aplicaţii, serverul şi planificatorul.*. In orice nod al reţelei în care se execută un task, se activează şi un exemplar de server.

O concluzie a subcapitolelor 1.3 şi 1.4 este aceea că există o tendinţă în dezvoltarea sistemelor de operare distribuite de a fi structurate ca un set de servere de sistem situate deasupra unui micro-kernel minimal (Amoeba, Chorus, Mach, Plan 9, Topaz, V-System); acest mod de structurare a componentelor sistemului de operare oferă suport pentru modulariatate, portabilitate, scalabilitate, dar şi adaptare la noi arhitecturi

BUPT

hardware în condiții de modificare rapidă a cerințelor pieții. În cadrul acestei arhitecturi, micro-kernelul furnizează cîteva funcții de bază, independente de particularitățile sistemului de operare, legate de alocarea procesorului și gestiunea memoriei precum și facilitatea de comunicare între procese (Inter-Process Communication-IPC). Astfel, se oferă severelor de sistem posibilitatea de a interacționa independent de mediul în care sunt executate, mediu care poate fi multiprocesor, multicomputer sau configurație de rețea.

Serverele de sistem furnizează, pe baza primitivelor puse la dispoziție de micro-kernel, un set de funcții specifice sistemului de operare, care realizează gestiunea resurselor fizice și logice (fișiere, dispozitive de I/O, linii de comunicație); un set de servere de sistem este uneori denumit ***subsistem***. Subsistemele, pe de o parte, separă funcțiile sistemului de operare în seturi de servicii furnizate de servere autonome, și , pe de altă parte, oferă seturi de API-uri programelor de aplicație.

Pentru asigurarea unui mediu de programare bazat pe standardul Unix, un susbsistem special poate fi rezervat pentru implementarea apelurilor sistem Unix. Serviciile Unix pot fi astfel extinse la medii distribuite, multiprocesor sau chiar timp real, într-o manieră transparentă pentru programele aplicative.

Arhitectura bazată pe un micro-kernel oferă, pe de altă parte și suport adecvat pentru dezvoltări ulterioare ale sistemului de operare. Omogenitatea interfețelor serverelor de sistem, datorată folosirii aceloraşi facilități IPC, oferite de micro-kernel, permite programatorilor de sistem să dezvolte noi servere pe care să le integreze ulterior în sistem. De exemplu, noi strategii de gestiune a fișierelor sau facilități de toleranță la defecte pot fi înglobate în noi servere, care întîi vor fi testate ca programe utilizator obișnuite, fără a afecta astfel funcționalitatea sistemului în ansamblu și integrate ulterior, după testarea lor, în sistem.

O altă concluzie a capitolului întîi este legată de instrumentele ce asistă proiectantul în dezvoltarea sistemului de operare distribuit; aceste instrumente implică și un anumit grad de abstractizare. Astfel, în proiectarea sistemelor de operare distribuite există două modele: ***modelul obiect*** și ***modelul proces***. Conceptul de obiect folosit în modelul obiect dispune de proprietățile de încapsulare și abstractizare. După cum obiectele active permit execuția paralelă a mai multor procese componente sau a unui singur proces, modelul obiect se divide în ***modelul obiecte multiactive*** și ***modelul obiecte active secvențiale***.

Capitolul 2. Comunicații în sisteme distribuite. Protocoale.

Arhitectura bazata pe un micro-kernel minimal furnizează, pe de o parte, o soluție de rezolvare într-o manieră structurată și modulară a problemei creșterii continue a complexității sistemelor de operare, dar, pe de altă parte, furnizează și avantajul unui cadru general pentru proiectarea, dezvoltarea și integrarea sistemelor "deschise". Termenul de arhitectură deschisă este tratat în capitolul 2 în sensul accepțiunii OSI: un sistem *deschis* este în acceptiunea ISO/OSI un sistem cu o arhitectură standardizată public, care poate să fie conectat și deci să comunice cu orice alt sistem deschis care utilizează același model de arhitectură. La baza conceptului de arhitectură deschisă stau două principii esențiale: (1) fiecare nivel valorifică serviciile asigurate de nivelele inferioare, culminînd cu nivelul aplicație, căruia îi sunt furnizate cît mai multe servicii pentru proiectarea de aplicații distribuite; (2) este asigurată independența fiecărui nivel, prin definirea serviciilor care trebuie asigurate nivelului următor, indiferent de modul de realizare al acestor servicii.

Există două direcții în cadrul cercetărilor asupra protocoalelor nivelului transport: (1) proiectarea și implementarea unor protocoale specifice unui sistem de operare distribuit ***(protocoale cu caracter restrîns)***; (2) reexaminarea proiectării și implementării ***protocoalelor cu caracter universal***. Pentru ambele tipuri de protocoale există argumente pro și contra. Astfel, *protocoalele cu caracter universal* oferă avantaje precum portabilitate, independența aplicațiilor, dar și dezavantaje care provin din ignorarea aspectelor specifice ce apar în sistemele distribuite; aceste dezavantaje constau într-o complexitate excesivă (protocoalele de transport la nivelul ISO/OSI sunt un exemplu în acest sens) cu repercursiuni asupra performanțelor, orientarea preponderentă pe variante de orientate pe conexiune, lipsa unui suport pentru implementarea unor facilități speciale în rețelele locale precum partajarea resurselor multicasting. *Protocoalele cu caracter restrîns* au ca principal avantaj performanțe mai bune, (întîrzieri mici, printr-o execuție eficientă) dar și dezavantajul aplicabilității numai într-un anumit tip de rețele si/sau arhitectură de aplicații.

Există o serie de cerințe pe care trebuie să le respecte un nou protocol Internet, rezolvate în totalitate de IPv6, motiv pentru care acesta a fost desemnat ca standard al noii generații de protocoale Internet: (1)*compatibilitate cu IPv4*; (2)-*facilități extinse în ceea ce privește posibilitățile de adresare și de rutare*. (3)*noi funcționalități* precum servicii real-time, autentificare, asigurarea securității, integrității și confidențialității datelor.

Capitolul 3. Paradigme ale interacțiunii dintre procese în sisteme distribuite.

Într-un sistem distribuit, o aplicație poate fi structurată în mai multe procese repartizate, pe mai multe procesoare, dispuse pe mai multe noduri într-o rețea. Cele mai cunoscute clase de primitive utilizate pentru comunicația între procese sunt: (1) *comunicarea prin mesaje*, care se asociază modelului client/server și la care fluxul de mesaje poate fi unidirecțional sau bidirecțional; (2) *apel de procedură la distanța (RPC)*, care oferă un mecanism puternic tipizat pentru transferul bidirecțional al informației; (3) *tranzacții*, care oferă un mecanism complex pentru executarea și sincronizarea de operații asupra unor obiecte partajate, asigurînd recuperarea informației în cazul apariției de erori hardware și software; (4) *comunicare prin conducte* (*pipes*)

BUPT

ca o generalizare a modelului conductelor locale introdus de sistemele Unix. Aceste clase de primitive permit implementarea a patru tipuri de procese într-un program distribuit: *filtre, clienti, serveri și perechi.*

Capitolul 4 prezintă problemele care apar la implementarea primitivelor din clasele 1,2 și 4. Modelul client-server este abordat atît sub aspectul concurenței la nivelul serverelor și clienților, al protocolului utilizat, orientat sau neorientat pe conexiune, cît și sub aspectul cerințelor impuse realizării unor sisteme client-server.

Comunicarea prin mesaje este asociată cu modelul client/server și constituie o extensie a comunicării locale interprocese prin mesaje; este o formă de comunicare în care procesele utilizator generează explicit mesajele transmise prin subsistemul de comunicație și utilizează explicit mecanismele de livrare și recepție ale mesajelor. Referitor la semantica primitivelor pentru comunicare, se separă primitivele în primitive *cu și fără blocare, sigure și nesigure, cu și fără buferare*; în capitolul 4 se arată, că, pentru primitiva *send*, această clasificare este generată de faptul că un program apelant al unei operații *send* poate obține controlul în următoarele 4 moduri: (1) semantica *no-wait-send* sau *asynchronous-send*- procesul emitator asteapta doar preluarea mesajului de catre subsistemul de comunicație;(2) semantica *syncronization send* - procesul emitator asteapta pînă cînd nodul destinație primește mesajul;(3) semantica *rendez-vous* - procesul emitator asteapta pînă cînd procesul receptor primește mesajul;(4) semantica *remote-invocation send* sau *remote procedure call* - procesul așteaptă pînă cînd primește un răspuns de la procesul receptor, în urma tratării cererii.

Modul de lucru cu buferare la nivelul programului utilizator, pentru primitiva *receive* , este exemplificat prin aplicația prezentată în anexa C4, în care mesajele primite de un server aflat pe un calculator dintr-o rețea Novell sunt memorate într-o coadă de mesaje, lansîndu-se o rutină ***EventServiceRoutine***.

Modelul de comunicație bazat pe *apel de procedură la distanța (RPC=Remote Procedure Call)* permite unui proces să apeleze și să execute o procedură de la distanță ca și pe o procedură locală. Pe timpul execuției procedurii de la distanță, apelantul este blocat, pînă cînd se receptionează rezultatul. Prin RPC se permite astfel extinderea apelului de proceduri, folosit în limbajele de programare la mediile distribuite. Modelul RPC este atractiv pentru că oferă un mecanism de comunicație în cadrul programelor distribuite, care se poate caracteriza astfel: (1) este simplu și familiar (datorita similarității cu apelurile de proceduri locale); (2) este general (apelul de procedură este folosit în toate aplicațiile); (3) poate fi implementat eficient. Modelul RPC simplifică construcția programelor distribuite, permițînd ignorarea de catre acestea a detaliilor de comunicație și a erorilor de transmisie. Sunt necesare cinci elemente pentru a se obține efectul execuției unei proceduri locale într-un mediu distribuit: *procesul client, codul stub pentru client, mecanismul de transport, codul stub pentru server și procesul server.* Modelul RPC poate fi interpretat și ca o extindere a primitivelor sigure, cu blocare, structurate: *Send, GetRequest, SendResponse.* Spre deosebire de aceste primitive, datorita simplității și totodată generalității sale, modelul RPC poate fi însă extins la mai multi clienți și la mai multe servere, ca și la medii eterogene.

Exemplificările pentru modelul RPC sunt realizate în capitolul 3 pentru DCE RPC, Microsoft RPC și Sun RPC; se arată cum se realizează legătura clienților la servere, concurența multifir la nivelul serverelor, dezvoltarea aplicațillor distribuite.

Capitolul 3 abordează și celelalte două tipuri de interacțiuni între procese în sisteme distribuite: comunicațiile de grup și comunicația la distanță prin ***pipe***. Un *grup multicast* este un set de procese care reprezintă destinația aceluiași mesaj. ***Comunicațiile de grup** (sau multicast)* se referă la comunicații de tipul unul-la-mai-multi; grupurile sunt entități dinamice; pot fi create și distruse grupuri, iar componența grupurilor poate fi modificată. În cadrul unui grup, procesele pot fi egale sau poate fi creată o ierarhie între procese; de exemplu, un proces poate indeplini rolul de coordonator, în timp ce celelalte pot îndeplini aceeași sarcină de lucru. Primul model prezinta o simetrie; dacă unul dintre procese dispare, grupul devine mai mic, dar continuă să lucreze. În cel de-al doilea, dispariția coordonatorului pune în pericol întregul grup. În schimb, în primul model, luarea unei decizii este o operație complicată, care implică un proces de votare. Pentru ***gestiunea grupurilor*** (crearea, stergerea grupurilor, modificarea componenței grupurilor) există două soluții: (1) o soluție ***centralizata***, în cadrul căreia un server de grupuri primește și tratează toate cererile care se refera la componența grupurilor; (2) o soluție ***distribuita***, în cadrul căreia un proces transmite un mesaj tuturor proceselor din toate grupurile la intrarea sa într-un grup, ca și la parasirea unui grup. Grupurile pot fi *inchise*, caz în care numai membrii unui grup pot transmite mesaje grupurilor sau *deschise*, caz în care orice proces din sistem poate transmite mesaje la orice grup.

Comunicarea între procese prin conducte (*pipes)* este un mecanism care permite transferul datelor între procese utilizînd strategia FIFO, concomitent cu sincronizarea proceselor implicate în transfer. Ea poate fi aplicată și modelului client server prezentat anterior deși nu îi este caracteristică. Microsoft Windows (Win32) furnizează pentru comunicația la distanță ***conducte cu nume***. În subcapitolul 3.3.2 se arată cum pot fi proiectate procesele cu rol de server într-o comunicație prin conducte astfel încît să funcționeze multifir; soluția este să se creeze mai multe instanțe ale unei conducte, iar fiecărei instanțe să i se asocieze un fir în cadrul căruia se așteaptă cereri de conectare și se deservesc cereri de servicii.

O concluzie care rezultă din cele prezentate în capitolul 3 ar fi următoarea: comunicarea prin mesaje, care este considerată o caracteristică definitorie a arhitecturii bazate pe un micro-kernel este cea mai naturală cale de a structura sistemele în care componenetele sunt distribuite pe un set de procesoare slab cuplate; ea oferă o izolare foarte clară între componentele sistemului, impunînd reguli de comunicație explicită între

BUPT

acestea, dar, în acelaşi timp, oferă şi o soluţie flexibilă pentru a asambla ierarhic componentele distribuite, în entităţi globale, situate pe nivele superioare. Comunicaţia între componente poate îmbrăca forma unor protocoale în care se folosesc primitivele *send/receive simple,* care asigură transferul de date, sau forma unor mesaje *send/reply,* care pot fi combinate într-o formă de RPC, pentru a deservi cît mai bine comunicaţiile de tip client-server.

Capitolul 4. **Sincronizarea în sisteme de operare distribuite.**

Comunicaţia între procese asigură nu numai transferul propriu zis al datelor ci şi sincronizarea execuţiei proceselor. Capitolul 4 este dedicat în întregime problemelor de sincronizare în sisteme distribuite. Datorită distribuirii resurselor, întîrzierilor de transmisie, lipsa unei memorii comune şi deci lipsa unor informaţii globale, algoritmii şi metodele pentru sincronizare din sistemele centralizate nu pot fi utilizate în sistemele distribuite. Există două clase mari de mecanisme de sincronizare în sistemele distribuite : ***centralizate*** şi ***distribuite***. Unele dintre aceste mecanisme se bazează pe conceptul de *excludere reciprocă* care presupune o ordonare a unor secvenţe de cod executate de procesele concurente; aceste mecanisme pot utiliza diferite tehnici precum: *un proces central coordonator* sau un *token circulant*. Alte mecanisme se bazează pe înregistrarea evenimentelor semnificative în cursul unor prelucrări asincrone; acestea utilizează obiecte precum *contori de evenimente* şi *secvenţiatori*, sau *semafoare distribuite*.

Capitolul 4 prezintă următorii algoritmi şi mecanisme pentru sincronizare: (1) *algoritmi distribuiţi pentru excludere reciprocă*: algoritmul lui Lamport, algoritmul lui Ricard şi Agrawala, algoritmul lui Rymond pentru excludere reciprocă în accesul la k resurse identice. Caracteristica comună a acestor algoritmi este faptul că se bazează pe mărcile de timp asociate cererilor de intrare în secţiunea critică şi pe schimb de mesaje; (2) *algoritmi care utilizează un token circulant* pentru excludere reciprocă: algoritmul pentru sisteme structurate în inel şi algoritmul lui Chandy, pentru sisteme nestructurate în inel precum şi o soluţie proprie a autorului pentru implementarea mecanismului de *secvenţiator circulant*; (3) *algoritmi broadcast* care folosesc facilitatea transmisiei broadcast. S-au prezentat: soluţia Andrews de implementare a *semafoarelor distribuite* precum şi soluţii proprii ale autorului pentru implementarea *contorilor de evenimente distribuiţi şi a secvenţiatorilor distribuiţi*. Prezentarea tuturor algoritmilor s-a făcut în mod unitar, utilizîndu-se pentru descrierea lor, descrierea SR introdusă de Andrew [And91]. Toţi algoritmii prezentaţi au fost implementaţi în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare; în anexele B1-B20 se prezintă şi codul sursă al implementărilor (din lipsă de spaţiu, se prezintă pentru fiecare algoritm numai o singură variantă: Unix, Microsoft Windows sau Novell NetWare). În anexa B20 se arată cum pot fi utilizaţi contorii de evenimente distribuiţi şi secvenţiatorii distribuiţi pentru a obţine excluderea reciprocă.

O serie de algoritmi şi mecanisme de sincronizare prezentaţi în acest capitol vor fi folosiţi în capitolele următoare pentru realizarea sicronizării în diferite aplicaţii. Astfel, pentru controlul accesului concurent la o bază de date în reţele locale Unix, Microsoft Windows (Win32) se folosesc semafoare distribuite; pentru implementarea unui toolkit pentru comunicaţii de grup, în scopul obţinerii accesului exclusiv în momentul alegerii unui identificator pentru un nou grup, se foloseşte o metodă cu token circulant în varianta în inel a implementării şi o metodă broadcast în varianta broadcast a implementării.

Capitolul 5. **Proiectarea şi implementarea unor mecanisme suport pentru dezvoltarea de aplicaţii în reţele Unix, Microsoft Windows (Win32) şi Novell NetWare.**

Prima parte a lucrării se încheie cu un capitol cu caracter pur aplicativ, în care se prezintă proiectarea şi implementarea a două toolkit-uri.

Subcapitolul 5.1. este dedicat în întregime ***proiectării şi implementării unui toolkit pentru controlul accesului concurent la o bază de date***. Sunt prezentate 3 soluţii cu semafoare şi o soluţie bazată pe un proces coordonator. Implementările au fost realizate în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare. În finalul capitolului 5.1, în paragraful 5.1.4. se prezintă o comparaţie a celor 4 soluţii propuse şi cîteva concluzii rezultate în urma implementărilor; dintre care se enumeră: (1) Toate soluţiile prezentate necesită implementarea unor variabile de stare; acestea pot fi implementate fie separat, în cadrul unui proces coordonator, fie pot fi simulate ca "variabile comune" prin asimilarea lor cu valorile unor semafoare distribuite; (2) Prin faptul că menţin o listă de procese, prima soluţie cu semafoare şi soluţia procesului central pot planifica procesele pentru execuţie după diverşi algoritmi; (3) Pe de altă parte, a doua şi a treia soluţie cu semafoare, negestionînd liste, sunt mai rapide; (4) Implementarea soluţiei bazată pe un proces coordonator se face avînd în vedere raportul dintre timpul necesar tratării unei cereri şi rata de sosire a cererilor, pentru a elimina posibilitatea apariţiei situaţiei de "sufocare" a serverului.

Subcapitolul 5.2. este dedicat ***proiectării şi implementării unui subsistem software*** (în sensul descris mai sus, acela de set de servere de sistem) pentru comunicaţii de grup, care, la nivelul utilizator, se materializează într-un toolkit. Sunt propuse 4 variante: o variantă *semidistribuită*, două variante complet distribuite, care folosesc *transmisie broadcast*, respectiv protocolul *two phase commit* şi o variantă *în inel*. În toate soluţiile se consideră că se cunoaşte numărul participanţilor în sistem pentru a se putea lua decizia dacă un mesaj a fost recepţionat sau nu de *toţi* membrii unui grup. În variantele 1 şi 3 trebuie cunoscută şi identitatea participanţilor (nodurilor). În varianta a 4 a trebuie cunoscută identitatea vecinilor din inel.

BUPT

În proiectarea şi implementarea celor două toolkit-uri din capitolul 5 s-au folosit concepte, algoritmi şi diverse observaţii prezentate în toate capitolule anterioare; astfel au fost folosite protocoalele TCP, UDP şi IPX prezentate în capitolul 2, s-a ţinut seama de observaţiile prezentate în subcapitolul 3.1.2. relativ la concurenţa la nivelul serverelor, s-au folosit semafoarele distribuite prezentate la 4.3.4., algoritmii pentru excludere reciprocă prezentaţi la 4.2. şi s-au folosit ambele variante de realizare a serverelor: *multiproces* sau *multifir* (capitolul 1 şi 3). Testele au fost realizate utilizînd sistemele de operare Linux 2.0, DEC Alpha OSF 1.3, Microsoft Windows NT 4.0 şi Windows 95 precum şi Novell NetWare 3.11.

Partea a II-a: CRITERII DE PROIECTARE A SISTEMELOR DE PROGRAMARE DISTRIBUITĂ BAZATĂ PE OBIECTE (SPDBO). PROIECTAREA UNUI SUPORT SOFTWARE BAZAT PE MODELUL TRANZACŢIILOR ATOMICE PENTRU SPDBO ÎN REŢELE UNIX ŞI WINDOWS.

Capitolul 6. Sisteme de programare distribuită orientată pe obiecte (SPDBO). Criterii de proiectare

Cea de-a doua parte a lucrării cuprinde suportul teoretic necesar pentru proiectarea sistemelor de programare distribuită bazată pe obiecte precum şi proiectarea şi implementarea în reţele locale a unui suport software pentru lucrul cu tranzacţii atomice în aplicaţii distribuite.

Abordarea acestei teme s-a făcut ca urmare a constatării că, în ultimii ani, se conturează tot mai accentuat tendinţa ca, serviciile de gestiune ale tranzacţiilor, tradiţional implementate în sistemele de gestiune a bazelor de date să fie implementate în sistemele de operare. Aceasta permite ca serviciile pentru controlul concurenţei şi pentru lucrul cu tranzacţii atomice să fie disponibile oricărui program de aplicaţie, nu numai clienţilor unui manager al unei baze de date. Exemple sunt sistemul de operare Novell NetWare care pune la dispoziţie funcţii pentru lucrul cu tranzacţii atomice şi pentru manipularea lock-urilor asupra fişierelor (*Synchronization Services*) şi sistemul distribuit de fişiere Quick Silver Distributed System[CPS93], care furnizează o interfaţă de acces similară cu cea a sistemului Unix, cu observaţia că toate operaţiile sunt realizate într-o manieră atomică; aceasta înseamnă că, de exemplu, un client poate şterge, rescrie un fişier sau redenumi un director şi apoi decide dacă modificările vor fi permanentizate sau nu. QDFS garantează că, fie toate modificările vor fi permanentizate, fie niciuna nu va fi realizată; aceasta garanţie se face chiar pentru situaţii de avarii în reţea (defecţiuni ale nodurilor, ale liniilor de comunicaţie).

Partea a doua a acestei lucrări abordează aspectul tranzacţiilor atomice din perspectiva execuţiei lor în cadrul unui sistem distribuit de obiecte. Un ***obiect*** este definit ca o entitate care încapsulează informaţii de stare (sau date) şi un set de operaţii sau proceduri care manipulează aceste date într-un mod controlat, astfel încît pot fi tratate, colectiv, ca o singură unitate. În capitolul 6 se tratează aspecte relative la componenţa şi structura obiectelor, ale gestionării obiectelor, interacţiunii dintre obiecte şi al reprezentării obiectelor în memorie. O concluzie care se poate enunţa în urma parcurgerii capitolului 6 este aceea că proiectarea sistemelor de programare distribuită este o sarcină complexă şi dificilă, care trebuie să ţină seama de o multitudine de caracteristici şi funcţii necesare aplicaţiilor, unele specifice numai unei clase de aplicaţii. Abordarea obiectuală simplifică însă proiectarea, ţinînd seama că obiectele servesc ca unităţi de protecţie, sincronizare, recuperare şi mobilitate.

Capitolul 7. Controlul concurenţei şi recuperarea obiectelor

Capitolul 7 este axat în principal pe două probleme: realizarea atomicităţii tranzacţiilor efectuate asupra obiectelor şi controlul concurenţei acţiunilor la nivelul obiectelor.

În legătură cu ***realizarea atomicităţii tranzacţiilor***, este prezentat mai întîi modelul tranzacţional, apoi protocolul *two-phase commit*, cu implementarea sa OSI/ISO, protocolul *write-ahead logging*, alături de tehnicile cunoscute pentru recuperarea obiectelor într-un sistem tranzacţional: *value-logging, redo-only şi undo-only logging*. În finalul subcapitolului 7.3., al cărui subiect este realizarea atomicităţii tranzacţiilor se prezintă structura unui sistem de recuperare de obiecte; el este alcătuit din: (1) un *manager de recuperare* al cărui rol este de a urmări evoluţia tranzacţiilor, şi de a lua deciziile adecvate conform tehnicii de recuperare alese, în timpul execuţiei acţiunilor tranzacţiei, ca şi la execuţia comenzilor *commit*, *abort*; (2) un *manager pentru log*, utilizat de managerul de recuperare şi de alte componente ale sistemului distribuit pentru a scrie articole în log; scrierea articolelor în fişierul log este cerută fie de managerul de recuperare, fie de sistem.

În legătură cu ***mecanismele de control ale concurenţei*** se arată că acestea trebuie să asigure următoarele: (1) consistenţa obiectelor; (2) realizarea fiecărei tranzacţii atomice într-un timp finit. În plus, un ***bun*** mecanism de control al concurenţei se caracterizează astfel: (1) permite execuţia paralelă a acţiunilor, în scopul satisfacerii cerinţelor de performanţă ale sistemelor; (2) implică prelucrări minore; (3) se execută cu performanţe satisfăcătoare chiar şi în reţele în care întîrzierile de transmisie sunt semnificative; (4) impune puţine constrîngeri în structura tranzacţiilor. Soluţia tradiţională de rezolvare a problemelor de concurenţă în accesul la un sistem de obiecte este prin serializarea *tranzacţiilor*. Problema consistenţei datelor în aplicaţiile tradiţionale de baze de date a fost redusă la testarea serializabilităţii planificărilor; acest lucru se datorează faptului că nu se cunosc constrîngerile de consistenţă: fiecare acţiune într-o tranzacţie se abstractizează printr-o operaţie *read* sau *write*. Problema obţinerii serializabilităţii poate fi descompusă în două subprobleme: *sincronizarea read-write* şi *sincronizarea write-write*, notate pe scurt sincronizările *rw* şi *ww*. Sincronizarea *rw* se referă la serializarea tranzacţiilor în aşa fel încît fiecare operaţie *read* citeşte aceeaşi valoare ca şi cînd

BUPT

tranzacţiile ar fi executate serial; sincronizarea *ww* se referă la serializarea tranzacţiilor în aşa fel încît ultima operaţie *write* a fiecărei tranzacţii lasă datele în aceeaşi stare ca şi execuţia serială. Respectarea serializărior *rw* şi *ww* generează o prelucrare consistentă. Astfel, deşi sistemul de gestiune al bazei de date nu dispune de informaţii asupra constrîngerilor de consistenţă specifice aplicaţiei, garantează consistenţa, permiţînd numai execuţii serializabile ale tranzacţiilor concurente. În capitolul 7 se clasifică mecanismele tradiţionale de rezolvare a controlului concurenţei în: mecanisme bazate pe blocarea cu *lock*-uri, mecanisme bazate pe *mărcile de timp* şi mecanisme pentru *controlul optimist* al concurenţei. Conceptul de serializabilitate al tranzacţiilor este propriu mecanismelor tradiţionale pentru controlul concurenţei. Dacă sunt disponibile informaţii semantice despre tranzacţii şi despre operaţiile lor pot fi permise şi planificări neserializabile, dar totuşi consistente. O soluţie pentru rezolvarea cerinţelor *tranzacţiilor lungi* este renunţarea la planificările serializabile şi folosirea unor planificări care se bazează pe informaţiile semantice relative fie la obiecte, fie la operaţiile specifice aplicaţiei. În subcapitolele 7.4.7. şi 7.4.8. se prezintă două posibilităţi de extindere a mecanismelor tradiţionale pentru controlul concurenţei: ***mecanismul blocării altruiste*** şi ***mecanismul validării instantanee***.

Capitolul 8. Controlul interblocarii obiectelor

Problemele care se pun în legatură cu controlul interblocării în sistemele distribuite sunt asemănătoare cu cele pentru sistemele centralizate. În sistemele distribuite, însă, datorită dispersiei informaţiilor pe mai multe noduri, interblocările sunt şi mai greu de prevenit, evitat şi detectat.

Utilizarea tranzacţiilor atomice facilitează procedura de distrugere şi reluare a proceselor implicate intr-o interblocare; în cazul tranzacţiilor atomice, acestea pot fi readuse în starea de înaintea începerii tranzacţiei, pe cînd reluarea unor procese simple de la o stare anterioară necesită mai multe complicatii.

Pentru a detecta şi rezolva eficient interblocarea, este necesară utilizarea unor informaţii suplimentare despre aplicaţii. Aceste informaţii se referă la tipul cererii de alocare a resurselor. În consecinţă, algoritmii de detecţie a interblocărilor se pot clasifica în funcţie de modelele folosite pentru cererile de alocare: *modelul AND, modelul OR, modelul AND-OR, modelul* $\binom{k}{n}$ *şi modelul fără nici o restricţie.*

Interblocările pot fi detectate şi rezolvate utilizînd algoritmi ***centralizaţi, ierarhici sau distribuiţi.*** Algoritmii de detecţie şi rezolvare a interblocărilor sunt elaboraţi avînd la bază următoarele concepte: transmiterea căilor grafului dependenţelor tranzacţiei, urmărirea unui mesaj de probă de-a lungul arcelor grafului dependenţelor tranzacţiei, activarea unei prelucrări cu difuzie şi stabilirea stării globale consistente a sistemului.

În capitolul 8 se prezintă din fiecare clasă de algoritmi (centralizaţi, ierarhici sau distribuiţi) cîte cel puţin un algoritm exemplu. Pentru clasa algoritmilor distribuiţi se prezintă un algoritm *edge-chasing* pentru detecţia interblocării în modele AND şi un algoritm *cu difuzie* pentru detecţia interblocării în modele OR.

În sistemele distribuite în care sunt implementate tranzacţii atomice şi timp global sunt folosiţi în practică doi algoritmi pentru ***prevenirea interblocărilor***. Ambii se bazează pe ideea asignării fiecărei tranzacţii a unei mărci de timp, în momentul în care începe. Fiecărei tranzacţii i se mai asignează şi o prioritate, care poate fi numărul de proces; aceasta urmează să fie folosită în cazul în care două tranzacţii au aceeaşi marcă de timp. Cei doi algoritmi sunt: (1) algoritmul *wait_die* care se bazează pe o tehnică nepreemptivă; (2) algoritmul *wound_die* care se bazează pe o tehnică preemptiva. Ambii algoritmi evită amînarea la infinit a tranzacţiilor, cu condiţia ca, la reluarea tranzacţiilor după abortare, să se păstreze aceeaşi marcă de timp.

Capitolul 9. **Proiectarea unui suport software pentru programare distribuită bazată pe obiecte şi pe trazacţii atomice (DOSTP)**

Există actualmente un interes crescut în metodologii şi tehnici cu caracter general care simplifică construcţia sistemelor software distribuite robuste şi eficiente. Acest capitol prezintă o astfel de metodologie bazată pe tranzacţii atomice şi include proiectarea unui suport software pentru programarea distribuită bazat pe obiecte (DOSTP). Tranzacţiile atomice micşorează dificultatea scrierii unor programe de aplicaţii distribuite prin simplificarea tratării defectiunilor nodurilor şi prin rezolvarea problemelor de concurenţă. Tranzacţiile atomice sunt, la ora actuală o paradigmă caracteristică programării distribuite; ele au fost incluse într-o serie de sisteme şi chiar limbaje orientate pe obiecte.

Sistemul DOSTP a fost proiectat astfel încît să respecte următoarele cerinţe: (1) simplitate de apel ale funcţiilor sistemului DOSTP la nivelul aplicaţiilor utilizator; (2) modularitate şi extensibilitate, în sensul ca pot fi introduşi în sistem noi manageri de obiecte printr-un efort de programare minim; (3) sistemul trebuie să ofere posibilitatea execuţiei în paralel a mai multor operaţii asupra unor obiecte diferite; (4) sistemul trebuie să ofere posibilitatea recuperării obiectelor în urma defecţiunii unuia sau mai multor noduri implicate într-o tranzacţie atomică; (5) sistemul trebuie să se execute în mod utilizator; (6) sistemul trebuie să ofere o comunicaţie sigură; (7) obiectele trebuie să fie portabile (cu modificari minime, absolut necesare) atît în reţele UNIX cat şi în reţele Microsoft Windows (Win32); (8) tranzacţiile să nu poată fi amînate la infinit sau restartate la infinit.

În ceea ce priveşte maniera de proiectare a sistemului DOSTP se fac următoarele observaţii: (1) DOSTP foloseşte *modelul integrat* ca arhitectură generală; (2)pentru execuţia tranzacţiilor atomice se crează

BUPT

cîte un proces agent pentru fiecare obiect de la distanţă implicat într-o tranzacţie; (3) DOSTP utilizează *varianta dinamică a obiectelor active*, iar localizarea obiectelor se face folosind soluţia *serverelor de nume distribuite*; (4) concurenţa la nivelul obiectelor se rezolvă printr-o *schemă optimistă*; (5)pentru recuperarea obiectelor, DOSTP utilizează tehnicile *value-logging şi redo-only logging;* (6) pentru evitarea restartărilor la infinit ale tranzacţiilor, în DOSTP s-a implementat un mecanism *wound-die.*

În capitolul 9 se prezintă în detalii arhitectura sistemului DOSTP şi proiectarea elementelor sale componente: serverele de comunicaţii, managerii de tranzacţii, managerii de obiecte, serverele de nume, managerul central, procesele utilizator care realizează tranzacţiile. Sistemul de recuperare al obiectelor nu este implementat în DOSTP ca proces distinct; el se compune din module plasate în managerii de obiecte şi în managerii de tranzacţii şi din fişierele lor *log* aferente.

Capitolul 10. Implementarea sistemului DOSTP în reţele Unix şi Microsoft Windows (Win32)

Capitolul 10 este dedicat în întregime descrierii implementării sistemului DOSTP în reţele locale UNIX şi reţele Microsoft Windows (Win32). Descrierea este făcută comparativ, deoarece există deosebiri între cele două implementări, generate în special de faptul că în Unix s-au implementat servere multiproces iar în Windows servere multifir şi de faptul că există diferenţe în lucrul cu semafoare şi memoria partajată între cele două sisteme. Un element important, care trebuie subliniat, este proiectarea obiectuală a managerilor de obiecte. Scopul principal în realizarea managerilor de obiecte este ascunderea detaliilor de conectare cu sistemul tranzacţional. Managerii de obiecte sunt realizaţi astfel încît ei comunică numai cu serverul de comunicaţie local, fără a avea cunostinţă dacă serviciile le-au fost solicitate local sau de la distanţă. Orice manager de obiecte care urmează să se integreze în sistemul DOSTP trebuie să implementeze următoarele cinci seturi de funcţii: (1) *funcţia de conectare la sistem;* (2) *funcţia de eliminare din sistem;* (3) *funcţia de lansare* a unei noi tranzacţii, la solicitarea unui client; (4) *funcţia de recepţionare/ transmisie* a pachetelor din zona din/în zona de memorie partajată. (5) *funcţiile necesare modului de lucru tranzacţional.* Deoarece funcţiile din seturile 1-4 trebuie să le asigure orice manager de obiect, o primă consecinţă este că acestea, o dată scrise, realizarea unui nou manager de obiecte poate fi facută fără a se cunoaşte detalii despre implementarea sistemului tranzacţional; trebuie implementate numai funcţiile din setul 5. A doua consecinţă este că un nou manager de obiecte poate fi testat separat, înainte de integrarea sa în sistemul tranzacţional. Soluţia implementării obiectuale este utilă din ambele puncte de vedere.

Se prezintă, în final, concluziile autorului în legătură cu aspectele critice şi caracteristicile fundamentale ale unei noi generaţii de sisteme de operare distribuite, aşa după cum reies din parcurgerea acestei lucrări:

(1) Se apreciază că trebuie să se ofere facilităţi pentru integrarea în sisteme deschise, pentru multiprocesare, şi distribuţie masivă;

(2) Un element decisiv îl constituie optimizarea comunicaţiilor interprocese şi interprocesoare şi distribuţia, transparentă pentru utilizatori, a resureselor;

(3) O cerinţă importantă este portabilitatea pe diverse arhitecturi hardware, de la sisteme uniprocesor şi multiprocesor (puternic sau slab cuplate) pînă la medii puternic distribuite, de la spaţii de adrese liniare, segmentate, pînă la arhitecturi cu memorie virtuală;

(4) Este necesar să se ofere suport pentru pentru configurarea dinamică a serverelor de sistem dependente de hardware;

(5) Este necesară posibilitatea distribuirii serviciilor sistemului de operare ;

(6) Este necesară respectarea compatibilităţii cu interfeţele standard ale sistemelor deschise;

(7) Trebuie să se ofere un mediu adecvat pentru dezvoltarea, depanarea, extinderea şi integrarea atît a aplicaţiilor cît şi a noilor componente ale sistemului de operare; în ultimul caz trebuie să fie posibilă testarea componentelor în mod de lucru utilizator urmată, ulterior, după depanare, de migrarea în mod sistem;

(8) Arhitectura bazată pe un micro-kernel minimal permite o construcţie modulară a sistemului de operare, într-o manieră în care componentele de sistem pot fi dezvoltate şi asamblate în diferite moduri respectînd platformele hardware actuale, pe de o parte şi cerinţele aplicaţiilor pe de altă parte, făcînd posibilă cooperarea şi interacţionarea elementelor de sistem prin intermediul liniilor de comunicaţie de mare performanţă;

(9) O importanţă mare o prezintă infrastructura oferită pentru lucrul cu fire, actualmente standardizat de POSIX. Se apreciază însă că este nevoie de un suport mai bun pentru lucrul cu fire în ceea ce priveşte limbajele de programare, depanarea programelor şi modificarea bibliotecilor în biblioteci multifir.

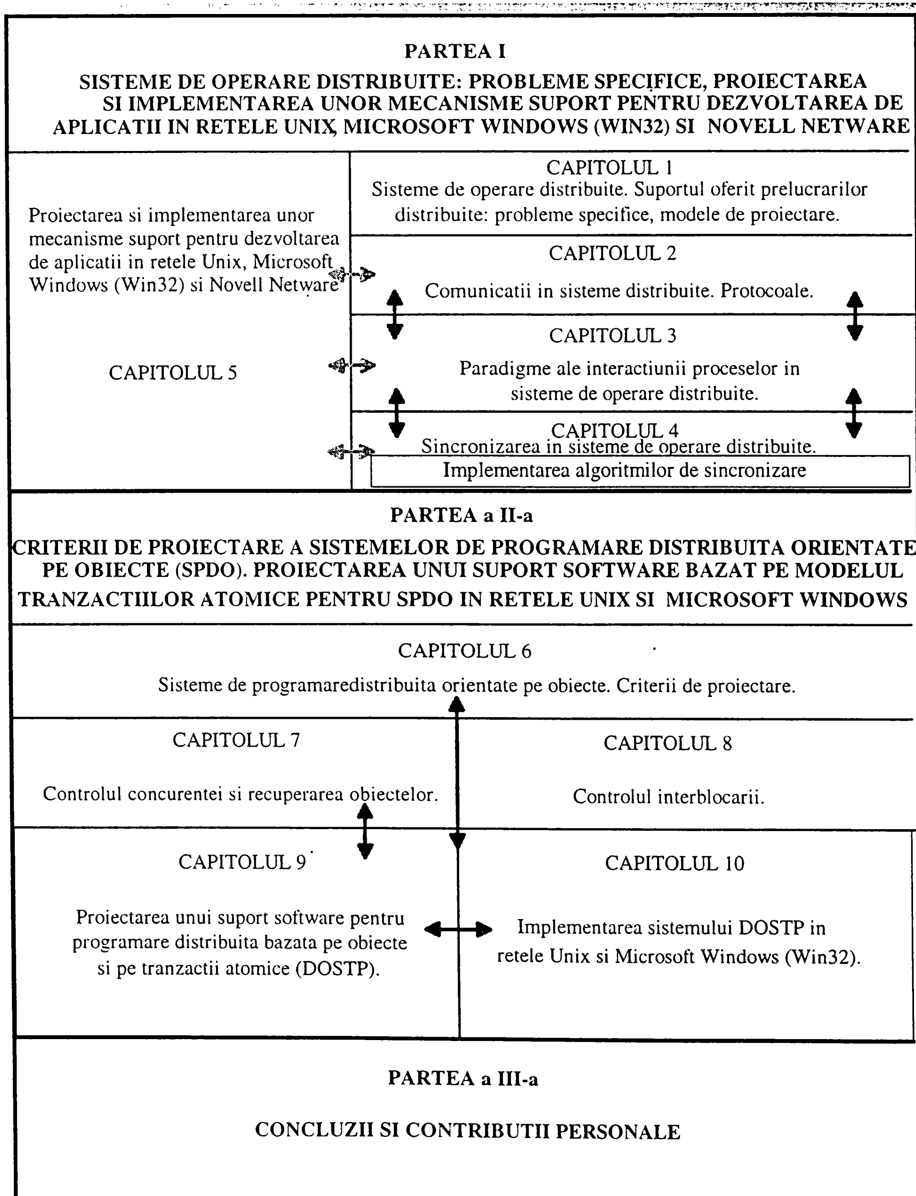
(10) În arhitecturile bazate pe obiecte, pentru a oferi suport adecvat aplicaţiilor, este necesară implementarea obiectelor partajate şi persistente, dotate cu controlul concurenţei şi posibilităţi de recuperare. Aceste cerinţe sunt realizate prin utilizarea conceptului de tranzacţie atomică, implementat prin protocolul *two-phase commit*, tehnicile de recuperare a obiectelor şi mecanisme de control al concurenţei (*optimistc sau pesimiste*).

(11) Conceptul de tranzacţie distribuită, caracterizat de proprietăţile ***ACID (atomicitate, consistenţă, integritate şi durabilitate***) poate constitui un suport software de mare ajutor pentru aplicaţii;

BUPT

(12) Sistemele distribuite cu facilități de toleranță la defecte necesită în plus față de serviciile obișnuite tip ***datagrame unicast, servicii de comunicații de grup,*** fie la nivel de procese, fie la nivel de obiecte.

Interdependența problemelor abordate în această lucrare și plasarea subsistemelor software dezvoltate de autor în raport cu noțiunile teoretice abordate sunt prezentate în figura următoare.



BUPT

11.2. Contribuţii originale, valoare aplicativă şi direcţii de dezvoltare

Pornind de la dublul scop al acestei lucrări - acela de a prezenta aspecte teoretice ale proiectării sistmelor de operare distribute şi de a proiecta şi implementa în reţele locale Unix, Microsoft Windows(Win32) şi Novell NetWare susbsisteme software - autorul a urmărit realizarea următoarelor obiective:

(A) selecţia şi sistematizarea unui mare volum de cunoştinţe din domeniul sistemelor de operare distribuite, atît din lucrările congreselor de specialitate din ultimii 15 ani cît şi din lucrările publicate în reviste de specialitate;

(B) elaborarea unor variante originale pentru: (a) proiectarea unui toolkit pentru comunicaţii de grup; (b)proiectarea unor mecanisme şi algoritmi pentru rezolvarea problemelor de concurenţă la nivelul obiectelor şi pentru recuperarea obiectelor într-un sistem de proiectare distribuită bazat pe obiecte şi tranzacţii atomice, şi găsirea unor soluţii de conectare automată a managerilor de obiecte la acest sistem; (c)proiectarea unor mecanisme pentru controlul accesului concurent la o bază de date.

(C) elaborarea unor variante originale de implementare în reţele locale a unor concepte precum contori de evenimente distribuiţi, secvenţiatori distribuiţi, şi implementarea în reţele locale Unix, Microsoft Windows(Win32) şi Novell NetWare a mai multor algoritmi cunoscuţi pentru sincronizarea proceselor în sisteme distribuite (algoritmi bazaţi pe marca de timp, algoritmi bazaţi pe un token circulant şi algoritmi bazaţi pe transmisie broadcast)

(D) implementarea în reţele locale Unix, Microsoft Windows(Win32) şi Novell NetWare a 3 susbsisteme software: (1) pentru comunicaţii de grup; (2) pentru controlul accesului concurent la o bază de date; (3) pentru proiectarea unui suport pentru un sistem de proiectare distribuită bazat pe obiecte şi tranzacţii atomice.

Avînd în vedere scopul şi obiectivele acestei lucrări se pot scoate în evidenţă următoarele contribuţii originale ale acestei teze:

(1) Prezentarea comparativă a noţiunilor de sistem de operare distribuit şi sistem de operare pentru reţele de calculatoare, prin prezentarea totodată a elementelor caracteristice fiecăruia.

(2) Expunerea unitară şi sistematică a problemelor referitoare la gestiunea proceselor şi gestiunea fişierelor în sisteme de operare distribuite.

(3) Abordarea şi prezentarea legăturii sistem de operare - prelucrare distribuită din 2 puncte de vedere: interconectarea programelor în reţele locale, pe baza unei descrieri a aplicaţiei (sistemul TDL, capitolul 1) şi punerea la dispoziţia utilizatorilor, într-un sistem de operare cunoscut (Unix) a unor API-uri pentru dezvoltarea de aplicaţii distribuite (DCE OSF).

(4) Prezentarea comparativă a 2 modele de proiectare a sistemelor de operare distribuite: ***modelul proces*** şi ***modelul obiect***, cu accentuarea submodelelor ***obiecte active secvenţiale*** sau ***obiecte multiactive*** ultimul fiind folosit de mai multe ori, pe parcursul acestei lucrări, pentru implementarea unor subsisteme.

(5) Sistematizarea cunoştinţelor legate de specificaţiile ISO/OSI în raport cu comunicaţiile în sisteme distribuite, alături de reliefarea rolului principal deţinut de nivelul transport în furnizarea de servicii pentru sistemele de operare.

(6) Exemplificarea protocoalelor prezentate în capitolul 2, protocolul IP - accentuîndu-se importanţa noii versiuni IPv6-, protocolul TCP, Novell IPX şi a comunicării prin socluri în Unix şi Microsoft Windows prin aplicaţii prezentate în anexele B şi C.

(7) Sistematizarea şi prezentarea unitară a aspectelor specifice ale suportului de comunicaţie pentru sisteme de operare distribuite, cu relizarea unei comparaţii între protocoalele cu caracter universal şi cele cu caracter restrîns.

(8) Prezentarea ***modelului client-server*** sub multiple aspecte: acela al concurenţei la nivelul serverelor, al protocoalelor specifice şi al sistemelor client-server.

(9) Expunerea completă a comunicării prin mesaje în sisteme distribuite, cu exemplificarea utilizării primitivelor *receive* fără blocare şi fără buferare (rezolvarea problemei buferării mesajelor la nivelul aplicaţiei) în aplicaţia din anexa C4.

(10) Prezentarea comunicaţiei prin conducte (***pipes***) la distanţă, cu exemplificarea serverelor multifir în Microsoft Windows NT 4.0.

(11) Prezentarea unitară a ***modelului RPC***, cu exemplificări pentru DCE OSF, Microsoft RPC sub multiple aspecte: transmiterea parametrilor şi rezultatelor, legarea clienţilor la servere, concurenţă, semantica RPC în prezenţa defectelor, protocoale RPC, extensii ale mecanismului de bază, aspecte critice în proiectare.

(12) Expunerea unitară a algoritmilor pentru sincronizarea proceselor în sisteme distribuite, utilizînd descrierea SR introdusă de G. Andrews.

(13) Prezentarea şi implementarea în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare a mai multor algoritmi pentru excludere reciprocă atît din clasa algoritmilor bazaţi pe marca de timp

BUPT

cît şi din clasa algoritmilor bazaţi pe transmiterea unui token circulant. Implementările algoritmilor sunt prezentate în anexele B1-B6.

(14) Elaborarea unor variante originale pentru implementarea unor mecanisme pentru sincronizarea proceselor în sisteme distribuite, precum contori de eveniment distribuiţi, secvenţiatori distribuiţi, secvnţiator circulant.

(15) Implementarea în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare a unor meacnisme pentru sincronizare precum: semafoarele distribuite, contori de eveniment distribuiţi, secvenţiatori distribuiţi, cu exemplificarea folosirii lor pentru obţinerea excluderii reciproce.

(16) Elaborarea mai multor variante pentru controlul accesului concurent la o bază de date în reţele locale de calculatoare şi comparaţia acestor variante.

(17) Implementarea în reţele locale Unix, Microsoft Windows (Win32) şi Novell NetWare a 4 soluţii pentru controlul accesului concurent la o bază de date în reţele locale: 3 soluţii bazate pe semafoare şi o soluţie bazată pe un proces coordonator. Implementările sunt prezentate în anexa C.

(18) Elaborarea a 4 variante originale pentru implementarea comunicaţiilor de grup: o variantă semidistribuită, o variantă cu transmisie broadcast, o variantă care foloseşte protocolul *two-phase commit* şi o variantă cu transmisie in inel.

(19) Implementarea în reţele Unix a 3 variante pentru comunicaţiile de grup: varianta semidistribuită (cap 5.2.2.), varianta cu transmisie broadcast şi varianta cu transmisie in inel (anexa A).

(20) Elaborarea unui studiu teoretic privind proiectarea sistemelor de programare distribuită bazată pe obiecte şi pe tranzacţii atomice, care tratează problemele de concurenţă la nivelul obiectelor, recuperarea obiectelor şi realizarea atomicităţii tranzacţiilor.

(21) Proiectarea unui suport software pentru programarea distribuită bazat pe obiecte şi pe tranzacţii atomice (DOSTP). Sunt prezentate detaliile de implementare ale fiecărei componente: serverele de comunicaţii, managerii de tranzacţii, managerii de obiecte, serverele de nume, managerul central, procesele utilizator care realizează tranzacţiile.. Se prezintă, de asemenea, circulaţia mesajelor în sistem în toate momentele distincte care pot apare în funcţionarea sistemului: conectarea unui nod la sistem, iniţierea unei tranzacţii, implicarea unui obiect local sau de la distanţă într-o tranzacţie, încheierea unei tranzacţii prin *commit* sau *abort,* recuperarea obiectelor după repunerea în funcţiune a unui nod coordonator sau neecoordonator al unei tranzacţii. Soluţia aleasă pentru implementare se bazeaza, în principal, pe două observaţii: (1) dezvoltarea unui sistem de programare distribuită poate fi simplificată prin utilizarea obiectelor, care devin entităţi autonome ce servesc ca unităţi pentru sincronizare, protecţie, recuperabilitate, securitate şi mobilitate. (2) gruparea acţiunilor asupra obiectelor în tranzacţii şi extinderea mecanismului tranzacţiilor la tipuri abstracte devine un instrument puternic în proiectarea unui astfel de sistem.

(22) Elaborarea unei metode de proiectare obiectuală a managerilor de obiecte atît pentru conectarea lor automată la sistemul tranzacţional cît şi pentru dezvoltarea şi testarea lor autonomă.

(23) Implementarea sistemului DOSTP în reţele locale Unix şi Microsoft Windows.

Aspectele prezentate în lucrare sunt rezultatele cercetărilor autorului din ultimii 7 ani, fiind valorificate în cadrul a 18 lucrări publicate la conferinţe ştiinţifice internaţionale şi în reviste de specialitate dintre care se remarcă articolul "The Design and the Implementation of a DOSTP- A Distributed Object Oriented System Based On Transactions Processing", publicat în revista *Facta Universitatis, series: Electronics and Energetics,* vol 9, no 2, 1996, pp 149-172, Nis, Yugoslavia, articolul "Object Oriented System for Distributed Transactions during the Development of a Product within a group of Companies", publicat la *1-st International Conference on Engineering Design and Automation,* Bangkok, Thailand, Martie 18-21, 1997 şi raportul de cercetare "Designing a Distributed Object Oriented System", publicat şi susţinut în programul TEMPUS prin două prelegeri la Technological Educational Institute of Piraeus, Faculty of engineering, Department of Computer Systems, Athens, Greece, mai 1996. Totodată, unele aspecte ale lucrării au putut fi valorificate într-un contract de cercetare ştinţifică încheiat cu Ministerul Învăţămîntului în anul 1994 (grant no 3006/14c/B9/1994) şi în activitatea didactică desfăşurată cu studenţii secţiei de calculatoare, Facultatea de Automatică, Calculatoare şi Electronică, Universitatea din Craiova.

Se remarcă caracterul aplicativ al rezultatelor acestei teze, pe viitor conturîndu-se noi perspective de valorificare, ţinînd cont de interesul crescînd care se manifestă faţă de sistemele distribuite.

BUPT

ANEXA A

Implementarea unui toolkit pentru comunicații de grup Managerul de grupuri pentru variantele distribuite

```
# Serverul pentru tratarea cererilor(managerul de grupuri) inglobeaza functiile serverului central si ale celui
# local din varianta semidistribuita prezentata la 5.2.2. si este identic pentru varianta cu transmisie broadcast
# si pentru varianta cu transmisie in inel

const MAX_NAME_LEN = ...            # lungimea maxima a numelor clientilor sau a grupurilor
const MAX_MES_LEN = ...             # lungimea maxima a unui mesaj
const MAX_CLIENTS_IN_GROUP = ...    # numarul maxim de clienti dintr-un grup
const MAX_MES_FOR_A_CLIENT = ...    # numarul maxim de mesaje care pot fi buferate pentru un client
const MAX_CONC_GROUPS = ...         # numarul maxim de grupuri la care poate apartine un client
const MAX_GROUPS = ..               # numarul maxim de grupuri
const MAX_CLI = MAX_CLIENTS_IN GROUP * MAX_GROUPS    # numarul maxim de clienti
const MAX_PROTO = ... # lungimea maxima a unui sir care indica protocolul folosit pentru autorizare clienti
const MAX_STRING = ... # lungimea maxima a unei parole folosita pentru autorizarea clientilor
const HEADER_LEN = ... # lungimea header-ului mesajelor
const MAX_ALIASES_FOR_A_GROUP = ...       # numarul maxim de pseudonime pentru un grup

type kind = enum (               # tipuri de mesaje care circula in sistem
        INIT_TGC, CLOSE_TGC, CREATE_GROUP, JOIN_GROUP, GET_ID_GROUP,
        LEAVE_GROUP,CREATE_ALIAS_GROUP, DESTROY_ALIAS_GROUP,
        SEND_TO_GROUP,RECEIVE_FROM_GROUP_NOW,GET_NO_OF_MESSAGES
        RECEIVE_FROM_GROUP_INFINITE, NO_MORE_RECEIVE_FROM_GROUP)
type error = enum (        # codificarea raspunsurilor la cererile trimise de clienti
        SUCCESS, MORE_MES, NO_WAIT_MESSAGES, ECLIENT_NOT_CONNECTED,
        ENO_AUTHORIZATION, ETOO_MANY_CLIENTS, ETOO_MANY_CONC_GROUPS,
        EFULL_GROUP, ECLIENT_ALREADY_CONNECTED ,
        ECLIENT_NOT_ IN_DEST_GROUP, EWRONG_SOURCE_GROUP,
        ECLIENT_NOT_ IN_SOURCE_GROUP, EWRONG_DEST_GROUP,
        ENO_BUF, EGROUP_ALREADY_CREATED, ETOO_MANY_GROUPS,
        EWRONG_ALIAS, EMES_TO_LONG, EBADF)

type mes_state = enum (free, in_transit, busy)        # starile unui slot in buferul de mesaje
type mes_to_group_manager = enum ( ACK, REQ) # tipul unui mesaj transmis server comunicatie->manager
                                             # de gestiune grupuri
type CLIENT_CONN = record of
        ( pid: int,                          # identificatorul unui client
          port: int,                         # identificatorul canalului rezervat dinamic unui client
          name[1:MAX_NAME_LEN]:char # numele clientului dat la conectare
        )                                    # structura care mentine identitatea clientilor
type MES_TO_GROUP = record of   # structura care mentine informatii relative la un mesaj receptionat
        ( mes[1:MAX_MES_LEN]: char,   # mesaj
          counter: int,                # nr proceselor care partajeaza mesajul
          source_gid: int,             # identificatorul grupului sursa
          dest_gid: int                # identificatorul grupului destinatie
        )
type GROUP_OF_CLIENTS = record of      # structura care mentine componenta unui grup
        (gid: int,                      # identificatorul de grup
         clients_no: int,               # numarul de clienti din acest grup
         pid_tab[1:MAX_CLIENTS_IN_GROUP]: int        # identificatori clienti
        )
type MES_TO_CLIENT = record of                # buffer de mesaje pentru un client
        ( pid: int,                           # identificatorul clientului
```


BUPT

```
        mes_no[1:MAX_MES_FOR_A_CLIENT]: int # indica numarul intrarii in buferul de mesaje
        pmes[1:MAX_MES_FOR_A_CLIENT]: pointer to MES_TO_GROUPS # indica mesajul
        pmes_state[MAX_MES_FOR_A_CLIENT]: mes_state)      # indica daca slotul este ocupat
        )
type WCLIENTS_FOR_GROUP = record of # structura care mentine identificatorii proceselor blocate,
                                    # in asteptare de mesaje de la un grup
        (gid: int,                  # identificatorul de grup
        wclients_no: int,           # numarul de procese in asteptare de mesaje de la grupul gid
        wpid_tab[1:MAX_CLIENTS_IN_GROUP]: int, # identificatorii proceselor in asteptare
        )
type CLIENTS_TO_GROUPS = record of  # structura care mentine grupurile la care apartine un client
        (pid :int                              # identificatorul clientului
        gid_no: int                            # numarul de grupuri la care apartine un client
        gid_tab[1:MAX_CONC_GROUPS]: int        # identificatorii grupurilor
        )
type MES_BUF = record of            # structura unei intrari in bufferul de mesaje
        (state: mes_state           # starea unei intrari: libera, ocupata, in tranzit
        message: MES_TO_GROUP)  # mesajul propriu zis transmis
type GROUP_ALIASES = record of      # structura care mentine pseudonimele grupului gid
        (gid: int,
        alias_no: int,              # numarul de pseudonime acordate grupului gid
        alias_tab[1:MAX_ALIASES_FOR_A_GROUP][1:MAX_NAME_LEN]: char # pseudonime
chan client_chan[1..MAX_CLI] (  # canal pentru comunicatie server local → client
        error_code: int ,           # codul erorii sau zero pentru succes
        dest_group_id: int,         # identificatorul grupului destinatie
        source_group_id: int,             # identificatorul grupului sursa
        len: int,                         # lungimea partii variabile a mesajului
        mes[1:len]: char )                # mesajul
chan local_chan[1..n] (   →     # canal pentru comunicatie client → server local
        knd: kind,              # tipul cererii
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int          # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int        # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                          # lungimea partii variabile a mesajului
        mes[1:len]: char)                  # mesajul
chan central_chan[1..n](        #  canal pentru comunicatie server local → server central
        knd: kind,              # tipul cererii
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int          # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int        # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                          # lungimea partii variabile a mesajului
        mes[1:len]: char)                  # mesajul
chan group_man_chan[1..n](      # canal pentru comunicatie server comunicatii → manager grupuri
        type: mes_to_group_manager ,       # ACK sau REQ
        err_code: int,                     # codul de eroare
        knd: kind,              # tipul cererii
        client_id: int          # identificatorul clientului de la care s-a trimis cererea
        dest_group_id: int          # identificatorul grupului de la care s-a trimis cererea
        source_group_id: int        # identificatorul grupului sursa care a trimis mesajul sau cererea
        group_name[1:MAX_LEN_GROUP]: char  # nume de grup
        len: int,                          # lungimea partii variabile a mesajului
        mes[1:len]: char )                 # mesajul
chan temp_chan (                # canal pentru conectarea unui client la serverul local
        error_code: int,  # codul erorii
        client_id: int)   # identificatorul clientului
```


BUPT

```
# informatii partajate cu managerul de grupuri  GM intr-o zona de memorie partajata
var no_groups: int                                              # numarul de grupuri
var group_aliases_tab[1:MAX_GROUPS]:GROUP_ALIASES               # mentine denumirile grupurilor
var group_of_clients_tab[1:MAX_GROUPS]:GROUP_OF_CLIENTS # mentine inf. despre componenta gr-lor
var mes_to_clients_tab: [1: MAX_CLI]:MES_TO_CLIENT  # mentine inf. despre mes. in asteptare pt.clienti
var messages_buf[1: MAX_MESSAGES]: MES_BUF              # bufferul de mesaje

# alte structuri de date
var client_conn_tab[MAX_CLI]: CLIENT_CONN              # mentine clientii activi si conexiunile lor
var wclients_for_group_tab[1:MAX_GROUP]:WCLIENT_FOR_GROUP# mentine identitatea clientilor blocati
var client_to_groups_tab[1: MAX_CLI]: CLIENT_TO_GROUPS # mentine grupurile tuturor clientilor
var knd:int, error_code:int,client_id:int, dest_group_id:int,source_group_id:int # folosite pentru citire mesaje
var mes_len: int, mes[1:MAX_MES]: char  # memoreaza lungimea si partea variabila a unui mesaj
var name[1: MAX_NAME_LEN]:char         # folosita pentru numele clientilor si grupurilor (alias-uri)
var auto_proto[1:MAX_PROTO]:char # folosita pentru citirea protocolului utilizat pentru autorizarea clientilor
var auto_string[1:MAX_STRING]: char                   # folosita pentru citirea unei parole
var message[1:MAX_MES]:char, source_gid: int          # variabile de  lucru
var pmessage: pointer to MES_TO_GROUP                 #variabila de lucru
var excl: semaphore := 1                              # folosit pentru accesul la memoria partajata
var start_GM: semaphore:=0             # pentru sincronizarea server comunicati - manager de grupuri

 initializari
 Down(start_GM)
 do true →
 if not empty(local_chan[i]) →
  receive local_chan[i] (knd, client_id, dest_group_id, source_group_id, name, mes_len , mes)
  if knd = INIT_TGC  and client_id ∉ clients_conn_tab →
   if knd = INIT_TGC →
      determina protocolul de autorizare client, auto_proto, si parola, auto_string din mes
       valideaza autorizarea clientului si pozitioneaza error_code
       if  error_code ≠ SUCCESS →
         send temp_chan(ENO_AUTHORIZATION,0)
      [] error_code  = SUCCESS →
         stabileste un identificator pentru client in client_id,
         ca fiind numarul primei intrari libere din client_conn_tab
         if client_id > MAX_CLIENTS → send temp_chan(ETOO_MANY_CLIENTS, 0)
         [] client_id ≤ MAX_CLIENTS →
                     creaza o intrare  clients_conn_tab[client_id]
                     Down(excl);creaza o intrare  mes_to_client_tab[client_id]; Up(excl)
                     send temp_chan (SUCCESS, client_id)
         fi
      fi
   [] knd = JOIN_GROUP  →
      if client_to_groups_tab[client_id].gid_no > MAX_CONC_GROUPS →
        send client_chan[client_id] (ETOO_MANY_CONC_GROUPS, dest_group_id, 0, 0, NULL)
      []client_to_groups_tab[client_id].gid_no ≤ MAX_CONC_GROUPS
         if dest_group_id ∈ group_of_clients_tab →
          if client_id  ∉  group_of_clients_tab[dest_group_id] →
             if group_of_clients_tab[dest_group_id].clients_no > MAX_CLIENTS_IN_GROUP →
                  send client_chan[client_id](EFULL_GROUP, dest_group_id, 0, 0, NULL)
             [] group_of_clients_tab[dest_group_id].clients_no ≤ MAX_CLIENTS_IN_GROUP →
                   Down(excl);insereaza client_id in group_of_clients_tab[dest_group_id];Up(excl)
                   insereaza dest_group_id in clients_to_group_tab
                  send client_chan[client_id](SUCCESS, dest_group_id, 0, 0, NULL)
             fi
         [] client_id  ∈  group_of_clients_tab[dest_group_id] →
           send client_chan[client_id](ECLIENT_ALREADY_CONNECTED, dest_group_id, 0, 0, NULL)
         fi
```


BUPT

```
            [] dest_group_id ∉ group_of_clients_tab →
                send client_chan[client_id](EWRONG_DEST_GROUP, dest_group_id, 0, 0, NULL)
            fi
        fi
    [] knd = LEAVE_GROUP →
        if dest_group_id ∉ group_of_clients_tab →
            send client_chan[client_id](WRONG_DEST_GROUP, dest_group_id, 0, 0, NULL)
        [] dest_group_id ∈ group_of_clients_tab →
            if client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
                Down(excl); sterge client_id din group_of_clients_tab[dest_group_id].pid_tab;
                fa mesajele i st mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id →
                    pmessage := mes_to_clients_tab[client_id].pmes[i]
                    mes_to_clients_tab[client_id].pmes[i] = NULL;
                    pmessage^.counter :=pmessage^.counter - 1        # un mesaj in asteptare mai putin
                    if pmessage^.counter = 0 →
                            deazaloca structura de la pmessage si intrarea
                                    mes_to_clients_tab[client_id].mes_no[i] din message_buf devine free
                    fi
                af
                Up(excl)
                send client_chan[client_id](SUCCESS, dest_group_id, 0, 0, NULL)
                if group_of_clients_tab[dest_group_id].clients_no = 0 →
                    send central_chan[i](LEAVE_GROUP, client_id, dest_group_id, 0, NULL, 0, NULL)
                fi
            [] client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
                send client_chan[client_id](ECLIENT_NOT_ IN_DEST_GROUP, dest_group_id, 0, 0, NULL)
            fi
        fi
    [] knd = CREATE_GROUP →
        if exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
                send client_chan[client_id](EGROUP_ALREADY_CREATED, 0, 0, 0, NULL)
        [] nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            if nu exista o intrare o intrare libera in group_of_clients_tab →
                send client_chan[client_id](ETOO_MANY_GROUPS,0, 0, 0, NULL)
            [] send central_chan[i](knd, client_id, 0, 0, name, 0, NULL)
            fi
        fi
    [] knd = GET_ID_GROUP →
        if nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            send client_chan[client_id] (EWRONG_ALIAS, 0, 0, 0, NULL)
        [] exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
            send client_chan[client_id] (SUCCESS, gid, 0, 0, NULL)
        fi
    [] knd = CREATE_ALIAS_GROUP →
        if dest_group_id ∉ group_of_clients_tab →
            send client_chan[client_id] (EWRONG_DEST_GROUP,dest_group_id, 0, 0, NULL)
        [] dest_group_id ∈ group_of_clients_tab →
            if exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
                send remote_chan[i] (CREATE_ALIAS_GROUP, EWRONG_ALIAS, client_id, 0,0,0,NULL)
            [] nu exista un grup, gid pentru care name ∈ group_aliases_tab[gid]→
                if group_aliases_tab[dest_group_id].alias_no > MAX_ALIASES_FOR_A_GROUP →
                    send remote_chan[i] (knd, ETOO_MANY_ALIASES,client_id,dest_group_id ,0,0,NULL)
                [] group_aliases_tab[dest_group_id].alias_no ≤ MAX_ALIASES_FOR_A_GROUP →
                    send central_chan[i] (knd, client_id, gid, 0, name, 0, NULL)
                fi
            fi
        fi
```


BUPT

```
[] knd = DESTROY_ALIAS_GROUP →
      if dest_group_id ∉ group_of_clients_tab→
           send client_chan[client_id] (EWRONG_DEST_GROUP, dest_group_id, 0, 0, NULL)
      [] dest_group_id ∈ group_of_clients_tab →
           if name ∉ group_aliases_tab[dest_group_id]→
                send client_chan[client_id] (EWRONG_ALIAS, dest_group_id, 0, 0,NULL)
           [] name ∈ group_aliases_tab[dest_group_id]→
                send central_chan[i] (knd, client_id, dest_group_id,0, name, 0,NULL)
           fi
      fi
[] knd = SEND_TO_GROUP →
      if source_group_id ∉ group_of_clients_tab →
         send client_chan[client_id](EWRONG_SOURCE_GROUP,    # se incearca o emisie dintr-un grup
                        dest_group_id,source_group_id,0,NULL)   # sursa inexistent
      [] source_group_id ∈ group_of_clients_tab →
        if client_id ∉ group_of_clients_tab[source_group_id].pid_tab →
           send client_chan[client_id](ECLIENT_NOT_ IN_SOURCE_GROUP,      # se incearca o emisie
                        dest_group_id, 0, 0, NULL)  # in contul unui client neconectat la sursa
        [] client_id ∈ group_of_clients_tab[source_group_id].pid_tab →
           if dest_group_id ∉ group_of_clients_tab →
           send client_chan[client_id](EWRONG_DEST_GROP, # se incearca o emisie la un grup destinatie
                                  dest_group_id,source_group_id,0,NULL)   # inexistent
          []formeaza la message un mesaj din adresa proprie IP, client_conn_tab[client_id].name, mes
             send central_chan[i](knd, client_id, dest_group_id, source_group_id, NULL,
                                                      mes_len + HEADER_LEN, message)
          fi
        fi
     fi
[] knd = RECEIVE_FROM_GROUP_INFINITE →
     if (source_group_id > MAX_GROUPS) or source_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](EWRONG_SOURCE_GROUP,     # se incearca receptia de la un grup
                        dest_group_id, source_gid, mes_len,message)  # inexistent
     [] (source_group_id ∈ MAX_GROUPS) →
        if dest_group_id ∉ group_of_clients_tab →
          send client_chan[client_id](EWRONG_DEST_GROUP, #se incearca o receptie in contul unui grup
                        dest_group_id, source_group_id,0, NULL)# care nu exista
        [] dest_group_id ∈ group_of_clients_tab →
           if client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
          send client_chan[client_id](ECLIENT_NOT_ IN_DEST_GROUP,  # se incearca o receptie
                        dest_group_id, 0, 0, NULL)  # in contul unui client neconectat la grupul destinatie
           [] client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
              if exista mesaje i in mes_to_client_tab[client_id] and
                           mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                           (source_group_id = ANY or
                           ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
                   Down(excl);
                   extrage primul mesaj i in message si calculeaza lungimea sa in mes_len
                    pmessage := mes_to_clients_tab[client_id].pmes[i]
                    mes_to_clients_tab[client_id].pmes[i] = NULL;
                    source_gid = pmessage^.source_gid
                    pmessage^.counter :=pmessage^.counter - 1      # un mesaj in asteptare mai putin
                   if pmessage^.counter = 0 →   deazaloca structura de la pmessage si intrarea
                         mes_to_clients_tab[client_id].mes_no[i] din message_buf devine free
                   fi
                   Up(excl);
                   if (mai exista mesaje i in mes_to_client_tab[client_id]) and
                              mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                              ((source_group_id = ANY ) or
```


BUPT

```
                        ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
                  send client_chan[client_id](MORE_MES, dest_group_id, source_gid,
                        mes_len, message) # se receptioneaza mesajul si o indicatie ca mai exista mesaje
                [] nu mai exista mesaje in asteptare de la source_group_id catre dest_group_id →
                  send client_chan[client_id](SUCCESS, dest_group_id, source_gid, mes_len,message)
                        #se receptioneaza mesajul si o indicatie ca nu mai exista mesaje
                fi
            []nu exista mesaje in asteptare de la source_group_id catre dest_group_id pentru client_id →
                if wclients_for_groups[source_group_id].wclients_no>MAX_CLIENTS_IN_GROUP →
                  send client_chan[client_id](ENO_BUF, dest_group_id, source_gid, mes_len,message)
                [] wclients_for_groups[source_group_id].wclients_no ≤ MAX_CLIENTS_IN_GROUP →
                  insereaza client_id in wclients_for_groups[source_group_id].wpid.tab
                  si actualizeaza wclients_for_groups[source_group_id].wclients_no
                fi
            fi
          fi
        fi
[] knd = NO_MORE_RECEIVE_FROM_GROUP →
      sterge client_id din wclients_for_groups[source_group_id].wpid.tab
      si actualizeaza wclients_for_groups[source_group_id].wclients_no
[] knd = RECEIVE_FROM_GROUP_NOW →
    if source_group_id > MAX_GROUPS or source_group_id ∉ MAX_GROUPS →
      send client_chan[client_id](EWRONG_SOURCE_GROUP,      # se incearca receptia de la un grup
                  dest_group_id, source_gid, mes_len,message) # inexistent
    [] (source_group_id ∈ MAX_GROUPS) →
      if dest_group_id ∉ group_of_clients_tab →
        send client_chan[client_id](EWRONG_DEST_GROUP #se incearca o receptie in contul unui grup
                  dest_group_id, source_group_id,0, NULL)# care nu exista
      [] dest_group_id ∈ group_of_clients_tab →
        if client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
          send client_chan[client_id](ECLIENT_NOT_ IN_DEST_GROUP,   # se incearca o receptie in
                  dest_group_id, 0, 0, NULL)  # in contul unui client neconectat la grupul destinatie
        [] client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
          if exista mesaje i in mes_to_client_tab[client_id] and
                      mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                      (source_group_id = ANY or
                      ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
              extrage primul mesaj i in message si calculeaza lungimea sa in mes_len
              Down(excl);
              pmessage := mes_to_clients_tab[client_id].pmes[i]
              mes_to_clients_tab[client_id].pmes[i] = NULL;
              source_gid = pmessage^.source_gid
              pmessage^.counter :=pmessage^.counter - 1      # un mesaj in asteptare mai putin
              if pmessage^.counter = 0 →   deazaloca structura de la pmessage si intrarea
                  mes_to_clients_tab[client_id].mes_no[i] din message_buf devine free
              fi
              Up(excl);
              if (mai exista mesaje i in mes_to_client_tab[client_id]) and
                          mes_to_clients_tab[client_id].pmes[i]^.dest_gid = dest_group_id and
                          ((source_group_id = ANY) or
                          ( mes_to_clients_tab[client_id].pmes[i]^.source_gid = source_group_id )) →
                send client_chan[client_id](MORE_MES, dest_group_id, source_gid,
                      mes_len, message) # se receptioneaza mesajul si o indicatie ca mai exista mesaje
              [] nu mai exista mesaje in asteptare de la source_group_id catre dest_group_id →
                send client_chan[client_id](SUCCESS, dest_group_id, source_gid, mes_len,message)
                      #se receptioneaza mesajul si o indicatie ca nu mai exista mesaje
              fi
          [] nu exista mesaje in asteptare de la source_group_id catre dest_group_id →
```


BUPT

```
                    send client_chan[client_id](NO_WAIT_MESSAGES, dest_group_id, source_gid,
                                                      0,NULL) #nu exista mesaje pentru receptie
                fi
              fi
            fi
          fi
 [] knd = GET_NO_OF_MESSAGES →
  if source_group_id > MAX_GROUPS or source_group_id ∉ MAX_GROUPS →
       send client_chan[client_id](EWRONG_SOURCE_GROUP,     # se incearca receptia de la un grup
                      dest_group_id, source_gid, mes_len,message)  # inexistent
   [] (source_group_id ∈ MAX_GROUPS) →
       if dest_group_id ∉ group_of_clients_tab →
         send client_chan[client_id](EWRONG_DEST_GROUP  #se incearca o receptie in contul unui grup
                      dest_group_id, source_group_id,0, NULL)# care nu exista
       [] dest_group_id ∈ group_of_clients_tab →
          if client_id ∉ group_of_clients_tab[dest_group_id].pid_tab →
         send client_chan[client_id](ECLIENT_NOT_ IN_DEST_GROUP,   # se incearca o receptie in
                      dest_group_id, 0, 0, NULL)   # in contul unui client neconectat la grupul destinatie
          [] client_id ∈ group_of_clients_tab[dest_group_id].pid_tab →
            contorizeaza in k mesajele i cu
                  mes_to_client[client_id].pmes[i]^.source_gid = source_group_id  si
                  mes_to_client[client_id].pmes[i]^.dest_gid = dest_group_id
               send client_chan[client_id](SUCCESS, dest_group_id, source_group_id, k, NULL)
         fi
       fi
   fi
   [] knd = CLOSE_TGC →
     sterge intrarea client_id din client_conn_tab[client_id]
     Down(excl);
     fa toate grupurile la care este conectat client_id, client_to_groups[client_id].gid.tab[i] →
       source_gid := client_to_groups[client_id].gid.tab[i]
       sterge client_id din group_of_clients_tab[source_gid].gid_tab[i]
              fa mesajele i st mes_to_clients_tab[client_id].pmes[i]^.dest_gid =source_gid →
                pmessage := mes_to_clients_tab[client_id].pmes[i]
                mes_to_clients_tab[client_id].pmes[i] = NULL;
                pmessage^.counter :=pmessage^.counter - 1          # un mesaj in asteptare mai putin
                if pmessage^.counter = 0 →  deazaloca structura de la pmessage  si intrarea
                      mes_to_clients_tab[client_id].mes_no[i] din message_buf devine free
                fi
             fa
      af
     Up(excl);
   fi
 fi
fi
if not empty(group_man_chan[i]) →
   receive group_man_chan [i](type, error_code, knd,
                              client_id, dest_group_id, source_group_id, mes_len, message)
   if knd = CREATE_GROUP →
       if error_code = SUCCESS →
           Down(excl); adauga o intrare in group_of_clients[dest_group_id]; Up(excl)
       fi
       if type = ACK →
           Down(excl)
           send client_chan[client_id](error_code, dest_group_id,0,0, NULL)
           insereaza dest_group_id in clients_to_group_tab[client_id]
           insereaza client_id in intrarea group_of_clients[dest_group_id]
           Up(excl)
```


BUPT

```
      fi
  [] knd = CREATE_ALIAS_GROUP
      if error_code = SUCCESS → adauga nume in group_aliases_tab[dest_group_tab] fi
      if type = ACK → send client_chan[client_id](error_code, dest_group_id,0,0, NULL) fi
  [] knd = DESTROY_ALIAS_GROUP
      if error_code = SUCCESS → sterge nume din group_aliases_tab[dest_group_tab] fi
      if type = ACK → send client_chan[client_id](error_code, dest_group_id,0,0, NULL) fi
  [] knd = LEAVE_GROUP →
      if error_code = SUCCESS →
      Down(excl);sterge intrarea dest_group_id din group_of_clients_tab   ; Up(excl);
      fi
  [] knd = SEND_TO_GROUP →
      if type = ACK → send client_chan[client_id](error_code, dest_group_id,0,0, NULL)
      if error_code= SUCCESS →
        if exista client_id astfel incit (client_id ∈ dest_group_id) and
                        (client_id ∉ wclients_for_group[source_group_id].wpid_tab) →
          Down(excl);
          se aloca o zona de memorie pentru o structura MES_TO_GROUP, in intrarea j din message_buf
           message_buf [j].state = busy; message_buf [j].message.mes := mes;
           message_buf [j].message.source_gid := source_group_id;
           message_buf [j].message.dest_gid := dest_group_id;
          se contorizeaza in k toti membrii grupului dest_group care nu sunt in
                                              wclients_for_group[source_group_id].wpid_tab
          message_buf [j].message.counter := k
          Up(excl)
       fi
       Down(excl);
       fa client_id ∈ dest_group_id st client_id ∉ wclients_for_group[dest_group_id].wpid_tab
          insereaza message_buf [j].message in
                 mes_to_client_tab[client_id].pmes si j in mes_to_client_tab[client_id].mes_no
       af
       Up(excl);
       fa (client_id ∈ dest_group_id) st (client_id ∈ wclients_for_group[source_group_id].wpid_tab)
          send client_chan[client_id](SUCCESS, dest_group_id, source_group_id, mes_len,mes)
                           #se receptioneaza mesajul si o indicatie ca nu mai exista mesaje
          sterge client_id din wclients_for_groups[source_group_id].wpid.tab
       af
      fi
     fi
   fi
  fi
od
```


BUPT

Anexa B
Sincronizare in sisteme distribuite

B1. Exemple de programe client care, pentru realizarea excluderii reciproce în rețele locale, folosesc un server care implementează unul din algoritmii prezentați în capitolul 4, la 4.3.2 și 4.3.3

```
/* ************************************************************ *
            Program client XWindows
  Foloseste functiile puse la dispozitie pentru un client
  de catre fiecare implementare a algoritmilor :

  int ExclAccessInitialize(void)
  void ExclAccessOpen(void)
  void  ExclAccessClose(void)
  void ExclAccessEnd());

************************************************************** */
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <X11/Xaw/List.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Dialog.h>
#include <X11/Xaw/AsciiText.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "exclu.h"
#define MAX_MES_LENGTH 100

Widget shell,box,quit,info,pane,req,rel,label;


void Quit(Widget w,XtPointer client_data,XtPointer call_data);
void Info(Widget w,XtPointer client_data,XtPointer call_data);
void Request(Widget w,XtPointer client_data,XtPointer call_data);
void Release(Widget w,XtPointer client_data,XtPointer call_data);


main(argc,argv)
int argc;
char **argv;
{
  XtAppContext app_cont;
  shell = XtVaAppInitialize(&app_cont, "XBox",
            NULL,0,&argc,argv, NULL,NULL);
  pane = XtVaCreateManagedWidget ("pane",
            panedWidgetClass, shell, NULL) ;
  box = XtVaCreateManagedWidget ("box",
            boxWidgetClass, pane, NULL);
  info = XtVaCreateManagedWidget("info",
            commandWidgetClass, box, NULL);
  req= XtVaCreateManagedWidget("request",
            commandWidgetClass, box, NULL);
  rel = XtVaCreateManagedWidget("release",
            commandWidgetClass, box, NULL);
  quit = XtVaCreateManagedWidget("quit",
            commandWidgetClass, box, NULL);
label = XtVaCreateManagedWidget("label",
            labelWidgetClass, pane,
            XtNlabel,"\n    Mutual Exclusion      \n",
            XtNwidth,400, NULL);
XtAddCallback(quit,XtNcallback,Quit,0);
XtAddCallback(info,XtNcallback,Info,0);
XtAddCallback(req,XtNcallback,Request,0);
XtAddCallback(rel,XtNcallback,Release,0);
if (ExclAccessInitialize())
  exit(1);
XtRealizeWidget(shell);
XtAppMainLoop(app_cont);
}

void Info(w,client_data,call_data)
/* apelata la actionarea butonului info */
/* afiseaza adresa nodului */
Widget w;
XtPointer client_data,call_data;
{
 char msg[MAX_MES_LENGTH];
 char *adr;
 if ((adr = GetServerAddress()) != NULL){
    sprintf(msg,"Host address= %s\n", adr);
    XtVaSetValues(label,XtNlabel,msg,NULL);
 }
 else {
 sprintf(msg," Unable to find the address\n");
 XtVaSetValues(label,XtNlabel,msg,NULL);
 }
}

void Request(w,client_data,call_data)
/* apelata la actionarea butonului request*/
/* realizeaza intrarea intr-o sectiune critica */
Widget w;
XtPointer client_data,call_data;
{
 char msg[MAX_MES_LENGTH];
 strcpy(msg," Waiting ...");
  XtVaSetValues(label,XtNlabel,msg,NULL);
  ExclAccessOpen();
 strcpy(msg," In critical section ... ");
  XtVaSetValues(label,XtNlabel,msg,NULL);
}

void Release(w,client_data,call_data)
/* apelata la actionarea butonului release*/
/* realizeaza iesirea dintr-o sectiune critica */
Widget w;
XtPointer client_data,call_data;
{
 char msg[MAX_MES_LENGTH];
  ExclAccessClose();
  strcpy(msg,"Done request...");
   XtVaSetValues(label,XtNlabel,msg,NULL);
}

void Quit(w,client_data,call_data)
/* apelata la actionarea butonului quit*/
/* asigura deconectarea de la server */
Widget w;
XtPointer client_data,call_data;
{
 ExclAccessEnd();
 exit(0);
}
```


BUPT

```
/* ***************************************************************** *
        Acelasi client dar in varianta Microsoft Windows

****************************************************************** */
#include <stdio.h>
#include <stdlib.h>
#include <winsock.h>
#include <windows.h>
#include "resource.h"
#include "generic.h"
#include "exclw.h"
#define MAX_STR 100

LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg,
WPARAM wParam, LPARAM lParam );
HINSTANCE hInst;
LPCTSTR lpszAppName = "MyApp";
LPCTSTR lpszTitle   = "Mutual Exclusion";

int APIENTRY WinMain( HINSTANCE hInstance,
 HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int
nCmdShow)
{
  MSG      msg;
  HWND     hWnd;
  WNDCLASS wc;
  WSADATA WSAData;
  UNREFERENCED_PARAMETER( lpCmdLine );
  UNREFERENCED_PARAMETER( hPrevInstance );
  if(GetVersion()& 0x80000000 && (GetVersion()&0xFF)==3){
      MessageBox( NULL,
          "This application cannot run on Windows 3.1.\n"
          "This application will now terminate.",
          "WinApp",
     MB_OK | MB_ICONSTOP | MB_SETFOREGROUND );
     return( 1 );
  }
  hInst = hInstance;
 wc.style          = CS_HREDRAW | CS_VREDRAW;
 wc.lpfnWndProc    = (WNDPROC)WndProc;
 wc.cbClsExtra     = 0;
 wc.cbWndExtra     = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon (hInstance, lpszAppName);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
  wc.lpszClassName = lpszAppName;
  RegisterClass( &wc );
  hWnd = CreateWindow( lpszAppName,
    lpszTitle,WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT, 0,CW_USEDEFAULT, 0,
   NULL,NULL, hInstance,NULL );
  if ( !hWnd )
     return( FALSE );
  ShowWindow( hWnd, nCmdShow );
  WSAStartup(MAKEWORD(1,1), &WSAData) ;
  if (ExclAccessInitialize()){
      MessageBox( NULL,
        "This application will now terminate .", "WinApp",
     MB_OK | MB_ICONSTOP | MB_SETFOREGROUND );
   exit(1);
   }
  while( GetMessage( &msg, NULL, 0, 0) )
  DispatchMessage( &msg );
  return( msg.wParam );
}

 LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg,
             WPARAM wParam, LPARAM lParam)
{
   HDC hdc;
  char str[MAX_STR], *adr;
  switch (uMsg)
  {
  case WM_CREATE:
     return DefWindowProc( hWnd, uMsg, wParam, lParam );
     break;
   case WM_DESTROY:
     PostQuitMessage( 0 );
     break;
   case WM_COMMAND:
     switch ( LOWORD( wParam ) )
     {
     case ID_INFO:
      SetWindowText( hWnd, "Server Address" );
      if ((adr=GetMyAddress()) != NULL){
        hdc = GetDC(hWnd);
       TextOut(hdc,1,1,"                  ",20);
       sprintf(str,"%s",adr);
       TextOut(hdc,1,1,str,strlen(str));
        ReleaseDC(hWnd,hdc);
     }
      break;
     case ID_REQUEST:
      SetWindowText( hWnd, "Waiting ...  " );
      ExclAccessOpen();
      SetWindowText( hWnd, "In critical section  " );
      break;
     case ID_RELEASE:
      ExclAccessClose();
      SetWindowText( hWnd, "Done!!! " );
      break;
    case ID_EXIT:
       DestroyWindow( hWnd );
       break;
     }
     break;
  default:
     return DefWindowProc( hWnd, uMsg, wParam, lParam );
   }
   return 0;
}

* **************************************************************** *
Functiile puse la dispozitia unui program client  de catre
 toti algoritmii  care realizeaza excluderea reciproca, prezenati
 in capitolul 4, la 4.3.2 si 4.3.3  -versiunea  Unix

****************************************************************** */
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "exclu.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define LOCAL_PORT   6561
#define FIFO_NAME  "enterChan"
#define MAX_HOST_NAME    50
#define MAX_MES_FIFO       6
#define LEN 400

int GetServerAddress(void);
int enterChan;       /* handler pipe */
int  lChan;
struct sockaddr_in sad;

int  ExclAccessInitialize (void)
{
 int errCode;
 if((enterChan= open(FIFO_NAME, O_RDWR))<0)
     return ERROR4;
  if ((errCode =GetServerAddress()) != 0)
    return errCode;
```


BUPT

```
bzero((char *)&sad, sizeof(sad)) ;
sad.sin_family     = AF_INET ;
sad.sin_addr.s_addr = inet_addr(MyServerAddress) ;
sad.sin_port       = htons(LOCAL_PORT) ;
if((lChan = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   return  ERROR5;
 return 0;
}

void ExclAccessOpen(void)
{
char info[LEN];
 sprintf(info,"%s","REQUEST");
sendto(lChan,info,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
read(enterChan,info,MAX_MES_FIFO);
}

void  ExclAccessClose()
{
 char info[LEN];
 sprintf(info,"%s","RELEASE");
 sendto(lChan,info,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
}

char * GetMyAddress(void)
{
 int errCode;
 if  ((errCode = GetServerAddress()) != 0)
    return NULL;
 else
   return MyServerAddress;
}

int GetServerAddress(void)
/*determina adresa serverului propriu*/
{
 struct in_addr adr;
 char HostName[MAX_HOST_NAME];
  struct hostent *host;

if (gethostname(HostName,MAX_HOST_NAME) != 0)
    return ERROR1;
 if((host = gethostbyname(HostName)) == NULL)
    return ERROR2;
 if(host != NULL)
     adr.s_addr = ((struct in_addr *)
(host->h_addr_list[0]))->s_addr;
 if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
   return ERROR3;
 return 0;
}

void ExclAccessEnd(void)
{
 unlink(FIFO_NAME);
 close(lChan);
}

/* ************************************************************ *
 Functiile puse la dispozitia unui program client  de catre
  toti algoritmii  care realizeaza excluderea reciproca, prezentati
  in capitolul 4, la 4.3.2 si 4.3.3  -versiunea  Windows

*************************************************************** */
#include <winsock.h>
#include <windows.h>
#include "client.h"
#include  "exclwerr.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define LOCAL_PORT   6561
#define MAX_HOST_NAME 50
#define LEN 400

int GetServerAddress(void);
 HANDLE enterChan;
 int    lChan;
 struct sockaddr_in sad;

int ExclAccessInitialize(void)
{
 int errCode;
 enterChan=OpenSemaphore(SYNCHRONIZE,TRUE,
                                   "LocalSem");
 if (enterChan==NULL)
      return ERROR4;
 if ((errCode = GetServerAddress()) != 0)
    return errCode;
 memset((void FAR *)&sad, 0, sizeof(sad)) ;
 sad.sin_family  = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(MyServerAddress) ;
 sad.sin_port  = htons(LOCAL_PORT) ;
 if((lChan = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    return ERROR5;
 return 0;
}
void ExclAccessOpen(void)
{
  char info[LEN];
 sprintf(info,"%s","REQUEST");
 sendto(lChan,info,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
 WaitForSingleObject(enterChan,INFINITE );
}
void ExclAccessClose()
{
 char info[LEN];
 sprintf(info,"%s","RELEASE");
 sendto(lChan,info,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
}

void ExclAccessEnd(void)
{
 closesocket(lChan);
}
char * GetMyAddress(void)
{
 int errCode;
 if  ((errCode = GetServerAddress()) != 0)
    return NULL;
 else
  return MyServerAddress;
}

int GetServerAddress(void)
/*determina adresa serverului propriu*/
{
 struct in_addr adr;
 char HostName[MAX_HOST_NAME];
  struct hostent *host;
if (gethostname(HostName,MAX_HOST_NAME) != 0)
    return ERROR1;
 if((host = gethostbyname(HostName)) == NULL)
    return ERROR2;
 if(host != NULL)
  adr.s_addr = ((struct in_addr *) (host->h_addr_list[0]))->s_addr;
 if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
   return ERROR3;
 return 0;
}

/* ************************************************************ *
 Program  demonstrativ pentru toti algoritmii de excludere
 reciproca  prezentati in capitolul 4. la 4.3.2 si 4.3.3  -versiunea
 Novell NetWare
- Programul utilizeaza functiile  :
    int ExclAccInitialize(void);
    int SetUsers(char users[MAX_CON][USER_LEN],
                          int hosts, int noObjects);
   void ExclAccEnd(void);
   void OpenExclAccess(void);
   void CloseExclAccess(void);
 puse la dispozitie de implementarea  fiecarui algoritm

*************************************************************** */
```


BUPT

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "exc.h"

void GetUsers (char users[MAX_CON][USER_LEN], int *hosts);
void GetObjects (int *noOb);
void ExitOnError (int val);
extern code;

void main(void)
{
  char c;
  char users[MAX_CON][USER_LEN];
  int hosts, noOb;
  int value, answer;
  clrscr();
  printf("Demonstration program for exclusive access:\n\n\
(1) Lamport algorithm\n(2) Ricard & Agrawala algorithm\n\
(3) Ray algorithm\n(4) Token ring algorithm\n(5) Token\
algorithm\n\n\
Choose an algorithm:");
  scanf("%d", &answer);
  do{
    GetUsers (users, &hosts);
    switch (answer){
     case 1:
     case 2:
     case 4:
     case 5:
            value = SetUsers(users,hosts,1);
            break;
     case 3:
            GetObjects(&noOb);
            value = SetUsers(users,hosts,noOb);
            break;
    }
    if (value) printf ("User %d not available! Try again.\n",value);
  }while (value);
  if ((value = ExclAccessInitialize()) != 0) ExitOnError(value);
  do{
    printf("Continue (Yes,No)?");
    c = getch();
    putchar('\n');
    if (c == 'n' || c == 'N'){
          ExclAccessEnd();
          printf("\nEnd demostration.\n");
          exit(0);
    }
    printf("Waiting to enter in critical section...\n");
    ExclAccessOpen();
    printf("In critical section...\n");
    c = getch();
    ExclAccessClose();
  }while (1);
}
void GetUsers (char users[MAX_CON][USER_LEN], int *hosts)
{
  int i;
  printf("\nNumber of processes: \n");
  scanf("%d",hosts);
  for(i=0;i<*hosts;i++){
    printf("User name for stations:");
    scanf("%s", users[i]);
  }
}
void GetObjects (int *noOb)
{
  printf("Number of objects = \n");
  scanf("%d",noOb);
}
void ExitOnError(int val){
 printf(" Eroare %d %d\n",val, code);
 ExclAccessEnd();
 exit(1);
}
```

B2. Implementarea algoritmului Lamport pentru obținerea excluderii reciproce - varianta Unix

```
/* **************************************************************** *
     Server care implementeaza  algoritmul   Lamport
   -Suportul teoretic este prezentat in capitolul  4, la 4.3.2.1.

****************************************************************** */
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdarg.h>
# include "Lampuer.h"

#define BROADCAST_ADDR "193.226.37.255"
#define REMOTE_PORT  6560
#define LOCAL_PORT 6561
#define BROADCAST_PORT  6562
#define FIFO_NAME "enterChan"
#define PERMS 0666
#define MAX_STRING    40
#define MAX_HOST_NO   20
#define MAX_HOST_NAME  256
#define MAX_TYPE_MES_LEN 20
#define LEN  400
#define TRUE      1
#define FALSE     0
# define DEBUG TRUE

/* *************************************************************** **
 *  Functii si variabile interne serverului                          *
 ************************************************************** ***/
typedef struct{        /* structura elementelor cozii de cereri */
  int  ts;             /*  marca de timp */
  int conn;            /* identificatorul host-ului */
}queue;
void CreateBroadcastChan(void);  /* creare canal broadcast */
int CreateRemoteChan(void);       /* creare  canal remoteChan*/
int CreateLocalChan(void);        /* creare  canal localChan */
int CreateClientChan(void);
void CreateEnterChan(void)        /* creare  canal enterChan*/;
void CloseEnterChan(void);
void FindAddress(void);          /* determina adresele host-urilor*/
void InOrder(queue qL[MAX_HOST_NO]);  /* ordoneaza coada*/
int max(int a,int b);
void Debug(const char *c, ...);
int enterChan ;          /* canal implementat prin FIFO */
int   bsd;               /*  soclu pt. transmisii broadcast */
struct sockaddr_in sad, cad,cla,sva,lCad;  /* adrese Inet */
int myId;                /* identificatorul host-ului local */char
connTab[MAX_HOST_NO][MAX_HOST_NO];
int processNo;           /* numarul de noduri participante la alg.*/
queue reqQueue[MAX_HOST_NO];  /* coada de cereri */        int
queueLength;             /* lungimea cozii de cereri */


void main (int argc, char *argv[] )
{
struct sockaddr_in sdr;            /* adresa Inet */
fd_set rfds;                       /* read file descriptor set*/
fd_set afds;                       /* active file descriptor set*/
int rChan,lChan,cli;               /* identificatorii canalelor */
int sval,lCadl;                    /* folcsite  la recvfrom */
```


BUPT

```
int sender, ts;          /* folosite pentru receptia mesajelor */
int LocalTS=0, HighTS = 0;  /* retin marcile de timp */
int inAccess = FALSE;       /*flag pentru intrarea in SC */
char sBf[LEN],rBf[LEN];     /* buffere */
char mes[MAX_TYPE_MES_LEN]; /* buffer */
int replyCounter = 0;  /* contorizeaza mesajele REPLY */
int nfds, err, i,j;         /* de lucru */

/* ***************************************************************** ***
faza de initializare: se prieia din linia de comanda identificatorul
host-ului local si numarul de procese.
Se creaza canalele.
Fiecare server trimite broadcast identificatorul sau si adresa sa
Se pun in asteptare de mesaje canalele remoteChan, localChan
***************************************************************** *** */

if(argc!=3) {
    printf ("Usage : [name] [identifier] [hosts] \n");
    exit(0);
}
myId = atoi(argv[1]);
processNo = atoi(argv[2]);
CreateBroadcastChan();
CreateEnterChan();
getchar();
FindAddress();
 for(i = 0;i<processNo;i++)
    Debug("Host= %d  of address= %s \n",i,connTab[i]);
rChan = CreateRemoteChan();
lChan = CreateLocalChan();
cli = CreateClientChan();
nfds = getdtablesize();
FD_ZERO(&afds);
FD_SET(rChan,&afds);
FD_SET(lChan,&afds);

/* ***************************************************************** ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
***************************************************************** *** */
while(TRUE){
Debug("Waiting messages..\n");
bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
if (select(nfds,&rfds,(fd_set *)0,(fd_set *)0,(struct timeval *)0)<0){
    Debug("Select error \n");
    exit(1);
}
if(FD_ISSET(rChan,&rfds)){  /* s-a primit un mesaj din retea */
  sval = sizeof(sva);
  err=recvfrom(rChan, rBf, LEN,0, (struct sockaddr*)&sva,&sval);
  if(err<0)
     Panic(ERROR14);
  scanf(rBf,"%2d%3d%s",&sender,&ts,mes);
  Debug("Received from %d  %s ts = %d...\n", sender,mes,ts);
  if(strcmp(mes,"REQUEST") == 0){  /* tratare REQUEST */
      reqQueue[queueLength].ts = ts;
      reqQueue[queueLength].conn= sender;
      queueLength++;
      InOrder(reqQueue);
      Debug("After reorder..\n");
      for(i=0;i<queueLength;i++)
          Debug("Q %dCon= %d ts =%d\n", i, reqQueue[i].conn,
                                          reqQueue[i].ts);
      HighTS = max(HighTS,ts);  /* actualizare ts */
      if(sender != myId){          /* transmite REPLY */
          bzero((char *)&sdr, sizeof(sdr)) ;
          sdr.sin_family   = AF_INET ;
          sdr.sin_addr.s_addr =  inet_addr(connTab[sender]) ;
          sdr.sin_port    = htons(REMOTE_PORT) ;
          sprintf(sBf,"%2d %3d %s",myId,1,"REPLY");
          Debug("Sending:%s\n",sBf);
          sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      }
  }
  else
  if(strcmp(mes,"REPLY") == 0){  /* tratare REPLY retea */
    replyCounter--;
    Debug("replyCounter = %d\n",replyCounter);
    if((replyCounter == 0) && (reqQueue[0].conn == myId)){
      inAccess = TRUE;          /* s-a obtinut accesul in CS */
      sprintf(sBf,"%s","enter");
      write(enterChan,sBf,6) ;    /* deblocare client */
    }
  }
  else
  if(strcmp(mes,"RELEASE") == 0){  /* tratare RELEASE retea */
   for(i=0;i<processNo;i++)
   if(reqQueue[i].conn == sender)
      for(j = i; j < queueLength-1; j++)
         reqQueue[j] =reqQueue[j+1];
   reqQueue[queueLength-1].ts = 0;
   reqQueue[queueLength-1].conn = 0;
   queueLength--;
   Debug(" RELEASE on remote , c nt=%d\n",replyCounter);
   if((replyCounter == 0) && (reqQueue[0].conn== myId)){
      inAccess = TRUE;          /* s-a obtinut accesul in CS */
      sprintf(sBf,"%s","enter");
      write(enterChan,sBf,6) ;  /* deblocare client */
   }
  }
}
if (FD_ISSET(lChan,&rfds)){ /* mesaj  de la clientul local */
 lCadl = sizeof(lCad);
err=recvfrom(lChan,rBf,LEN,0,(struct sockaddr*)&lCad,&lCadl);
  if(err<0)
      Panic(ERROR15);
  sscanf(rBf,"%s",mes);
  Debug("Received from my client  mes= %s\n",mes);
  if(strcmp(mes,"REQUEST") == 0){ /* tratare REQUEST local*/
    LocalTS = HighTS + 1;            /* pregatire ts */
    replyCounter = processNo-1;
    Debug(" ReplyCounter = %d \n", replyCounter);
    for(i=0;i<processNo;i++){   /* transmite cererea in retea */
       bzero((char *)&sdr, sizeof(sdr)) ;
       sdr.sin_family   = AF_INET ;
       sdr.sin_addr.s_addr = inet_addr(connTab[i]);
       sdr.sin_port    = htons(REMOTE_PORT) ;
       sprintf(sBf,"%2d %3d %s",myId,LocalTS,"REQUEST");
       Debug("Sending:%s la %d\n",sBf,i);
       sendto(cli,sBf,LEN,0,(struct  sockaddr*) &sdr,sizeof(sdr));
    }
  }
  else
  if(strcmp(mes,"RELEASE") == 0){  /* tratare RELEASE local*/
    inAccess = FALSE;
    for(i=0;i<processNo;i++)
      if(reqQueue[i].conn == myId)
        for(j = i; j < queueLength-1; j++)
          reqQueue[j] =reqQueue[j+1];
    reqQueue[queueLength-1].ts = 0;
    reqQueue[queueLength-1].conn= 0;
    queueLength--;
    for(i=0;i<processNo;i++)      /* transmite mesajul in retea */
      if(i != myId){
        bzero((char *)&sdr, sizeof(sdr)) ;
        sdr.sin_family   = AF_INET ;
        sdr.sin_addr.s_addr = inet_addr(conTab[i]);
        sdr.sin_port    = htons(REMOTE_PORT) ;
        sprintf(sBf,"%2d %3d %s",myId,1,"RELEASE");
        Debug("Sending:%s la %d\n",sBf,i);
        sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      }
   }
  }
}
}
/* ************************************************************* **
*                     Functii folosite de server                   *
************************************************************** ***/
void InOrder(queue qL[MAX_HOST_NO])
/* ordoneaza coada de cereri dupa marcile de timp */
{
queue mx;
int sort, i;
```


BUPT

```
 do{
   sort = TRUE;
   for(i=0;i<queueLength-2;i++)
     if(qL[i].ts > qL[i+1].ts){
               mx= qL[i];
               qL[i] = qL[i+1];
               qL[i+1] = mx;
               sort = FALSE;
      }
 }while(!sort);
}

void CreateBroadcastChan(void)
/* creaza canalul pentru transmisii broadcast  */
{
 int i;
 bzero((char *)&sad, sizeof(sad)) ;
 sad.sin_family   = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
 sad.sin_port    = htons(BROADCAST_PORT) ;
 if((bsd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR1);
 i = 1 ;
 if(setsockopt(bsd, SOL_SOCKET, SO_BROADCAST, (char *)
                                         &i, sizeof(i))) {
    Panic(ERROR2);
 }
  bzero((char *)&cad, sizeof(cad)) ;
  cad.sin_family   = AF_INET ;
  cad.sin_addr.s_addr = htonl(INADDR_ANY) ;
  cad.sin_port    = htons(BROADCAST_PORT) ;
  if(bind(bsd, (struct sockaddr *) &cad, sizeof(cad)) < 0)
    Panic(ERROR3);
  }

void FindAddress(void)
/* memoreaza in tabela de adrese adresele si identificatorii */
/* tuturor host-urilor                                        */
{
  struct in_addr  adr;
  struct sockaddr cadr;
  struct hostent *host;
   int cadrl;
   char HostName[MAX_HOST_NAME];
   char sBf[LEN];
  char code[MAX_STRING]; char string[MAX_STRING];
   int idRec,i;
  if (gethostname(HostName,256) != 0)
      Panic(ERROR4);
   if((host = gethostbyname(HostName)) == NULL)
      Panic(ERROR5);
   if(host != NULL)
     adr.s_addr = ((struct in_addr*)(host->h_addr_list[0]))->s_addr;
   if(strcpy(code,inet_ntoa(adr)) == NULL)
       Panic(ERROR6);
   strcpy(conTab[myId], code);
   sprintf(sBf,"%2d %s",myId,code);
   sendto(bsd,sBf,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
   for(i=0;i<processNo;i++){
      cadrl = sizeof(cad) ;
      recvfrom(bsd, sBf, LEN, 0, (struct sockaddr*)&cadr, &cadrl) ;
      sscanf(sBf,"%2d%s",&idRec,string);
      if(idRec != myId)
         strcpy(connTab[idRec],string);
    }
 }

 int CreateRemoteChan(void)
 /* creaza canalul pentru transmisii in retea  */
 {
  int s;
  if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     Panic(ERROR7);
  bzero((char *)&sad, sizeof(sad)) ;
  sva.sin_family  = AF_INET ;
  sva.sin_addr.s_addr = htonl(INADDR_ANY) ;
  sva.sin_port    = htons(REMOTE_PORT) ;
  if (bind(s, (struct sockaddr *)&sva, sizeof(sva)) < 0)
     Panic(ERROR8);
  return s;
 }
 int CreateLocalChan(void)
/* creaza canalul pentru receptia mesajelor de  la clientul local */
{
 int s;
  if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     Panic(ERROR9);
 bzero((char *)&lCad, sizeof(lCad)) ;
 lCad.sin_family = AF_INET ;
  lCad.sin_addr.s_addr = htonl(INADDR_ANY) ;
  lCad.sin_port    = htons(LOCAL_PORT) ;
  if(bind(s, (struct sockaddr *)&lCad, sizeof(lCad)) < 0)
      Panic(ERROR10);
  return s;
}
 int CreateClientChan()
/* creaza canalul pentru emisia mesajelor  in retea */
{
 int s;
  if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     Panic(ERROR11);
  bzero((char *)&cla, sizeof(cla)) ;
  cla.sin_family   = AF_INET ;
  cla.sin_addr.s_addr = htonl(INADDR_ANY) ;
  cla.sin_port    = htons(REMOTE_PORT) ;
  return s;
}

int max(int a,int b)
{
 if(a > b) return a ;
 else return b ;
}
void CreateEnterChan(void)
{
  unlink(FIFO_NAME);
  if( mknod(FIFO_NAME, S_IFIFO | PERMS , 0)<0)
     Panic(ERROR12);
   if((enterChan = open(FIFO_NAME, O_RDWR))<0)
     Panic(ERROR13);
}
void CloseEnterChan(void)
{
  unlink(FIFO_NAME);
}

void Panic( int err){
 printf("Exit:  %d error \n", err);
 exit(0);
}
void Debug(const char *c, ...){
  va_list args;
  if (DEBUG){
  va_start(args,c);
  vprintf(c, args);
  va_end(args);
   }
}
```


BUPT

B3. Implementarea algoritmului Ricard si Agrawala pentru obținerea excluderii reciproce varianta Novell NetWare

```
/* ************************************************************ *
         Program  care implementeaza algoritmul
                  Ricard si Agrawala
    -Suportul teoretic este prezentat in capitolul 4, la 4.3.2.2

************************************************************** */
#include <nxt.h>
#include <nit.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <mem.h>
#include <string.h>
#include "exclacc.h"

#define TRUE 1
#define FALSE 0
#define MAX_NODES 30
#define DATA_LENGTH 80
#define MAX_CON 30


/* ************************************************************ **
 *                 Functii si variabile folosite               *
 ************************************************************ ***/

int Initialize(void);     /* asigura initializare IPX */
int InitReceive(void);/* asigura  initializarea unui ECB pt. emisie*/
int InitSend(void);  /* asigura initializarea unui ECB pt. receptie*/
int GetConnectionNumbers(char sers[MAX_CON][USER_LEN]);
void far _loadds ReceiveESR(void);  /* rutina ESR */
void SendRequest(int serverConnNo, char* message);
void SendReply(IPXHeader *ppacketHeaderRec, char *message);
void RicAgrawRequest(void);
void RicAgrawEnter(void);
void RicAgrawRelease(void);
void ReceiveRequest(char *message);
void ReceiveReply(void);
void RicAgrawClose(void);

int noNodes;          /* numarul de statii */
int localTS;          /* valori pt. marcile de timp */
int highTS=0;
int replyCounter;     /* contorizeaza mesajele de reply */
int request;
int laterTab[MAX_NODES];
int inSection;        /* indica intrarea intr-o sectiune critica */
int later;
int noOb;

WORD connectionList[MAX_CON];
IPXHeader packetHeaderRec;
char packetDataSnd[DATA_LENGTH];  /* buffere */
char packetDataRec[DATA_LENGTH];
char packetDataReply[DATA_LENGTH];
char bufferSnd[DATA_LENGTH];
ECB ecbEmission;
ECB ecbReception;
WORD socketDest=0x5000;
WORD socketSnd=0x6000;
WORD socketRec = 0x5000;
int timeStamp, sender;
int code;

/* ************************************************************ **
 *               Functia de conectare la sistem                 *
 ************************************************************ ***/
int ExclAccessInitialize(void)
{
  int val;

  if ((val = Initialize()) != 0) return val;
  if ((val = InitSend()) != 0) return val;
  if ((val = InitReceive()) != 0) return val;
  return 0;
}

int SetUsers(char users[MAX_CON][USER_LEN], int hosts, int
noObjects)
{
  int value;
  noNodes = hosts;
  value = GetConnectionNumbers(users);
  if (value != 0) return value;
  noOb = noObjects;
  return 0;
}

int GetConnectionNumbers(char users[MAX_CON][USER_LEN])
{
  int i=0;
  WORD numberOfConnections = 0;
  WORD maxConnections = 20;
  for(i=0;i<noNodes;i++){
     GetObjectConnectionNumbers(users[i], OT_USER,
 &numberOfConnections, &connectionList[i], maxConnections);
     if(numberOfConnections == 0) return i;
  }
  return 0;
}

/* ************************************************************ **
 *  Functia pentru obtinerea accesului in sectiunea critica     *
 ************************************************************ ***/
void ExclAccessOpen()
{
  RicAgrawRequest();
  RicAgrawEnter();
}

/* ************************************************************ **
 *  Functia pentru deconectarea de la sistem                    *
 ************************************************************ ***/
void ExclAccessEnd(void)
{
  IPXCancelEvent(&ecbReception);
  IPXCloseSocket(socketRec);
  IPXCloseSocket(socketSnd);
}

/* ************************************************************ **
 *  Functia pentru iesirea din sectiunea critica                *
 ************************************************************ ****/
void ExclAccessClose(void)
{
 int j;
 disable();
 request = FALSE;
 for(j=1;j<noNodes;j++){
   if(laterTab[connectionList[j]-1])
            SendRequest(connectionList[j], "Reply");
   laterTab[connectionList[j]-1] = FALSE;
   }
 inSection = FALSE;
 enable();
}

int Initialize(void)
{
 if (( code = IPXInitialize()) != 0) return 1;
 return 0;
```


BUPT

```
}

int InitSend(void)
{
 if ((code = IPXOpenSocket((BYTE*)&socketSnd,0x00)) != 0)
     return 2;
 return 0;
}

int InitReceive(void)
{
 int code;

 /* initializarea structurilor de date */
 ecbReception.ESRAddress = ReceiveESR;
 ecbReception.fragmentCount = 2;
 ecbReception.socketNumber = socketRec;
 ecbReception.fragmentDescriptor[0].address =
&packetHeaderRec;
 ecbReception.fragmentDescriptor[0].size = sizeof(IPXHeader);
 ecbReception.fragmentDescriptor[1].address = (BYTE *)
packetDataRec;
 ecbReception.fragmentDescriptor[1].size = DATA_LENGTH;

 /* deschiderea soclului si intrarea in "ascultare" de pachete */
 if ((code = IPXOpenSocket((BYTE*)&socketRec,0x00)) != 0)
return 3;
 IPXListenForPacket(&ecbReception);
 IPXRelinquishControl();
 return 0;
}

void far _loadds ReceiveESR(void)
{
   disable();
   if (ecbReception.completionCode == 0){
     if(strcmp(packetDataRec, "Reply") == 0)
             ReceiveReply();
     else
             ReceiveRequest(packetDataRec);
   }
   enable();
   IPXListenForPacket(&ecbReception);
}

void SendRequest(int serverConnNo, char* message)
 {
   BYTE serverNode[6];
   BYTE serverNetwork[4];
   IPXHeader packetHeader;
   WORD socketDest=0x5000,socket;

   GetInternetAddress(serverConnNo,(char *)
           serverNetwork,(char *)serverNode,&socket);
   movmem(serverNetwork, packetHeader.destination.network,4);
   movmem(serverNode, packetHeader.destination.node,6);
   movmem(&socketDest, packetHeader.destination.socket,2);
   movmem(serverNode, ecbEmission.immediateAddress,6);
   movmem(message, packetDataSnd, DATA_LENGTH);
   ecbEmission.ESRAddress = NULL;
   ecbEmission.fragmentCount = 2;
   ecbEmission.socketNumber = socketSnd;
   ecbEmission.fragmentDescriptor[0].address = &packetHeader;
   ecbEmission.fragmentDescriptor[0].size = sizeof(IPXHeader);
   ecbEmission.fragmentDescriptor[1].address = (BYTE *)
                    packetDataSnd;
   ecbEmission.fragmentDescriptor[1].size = DATA_LENGTH;
   while (ecbEmission.inUseFlag != 0) IPXRelinquishControl();
   IPXSendPacket(&ecbEmission);
 }

 void SendReply(IPXHeader *ppacketHeaderRec, char *message)
 {
   IPXHeader packetHeader;
   movmem(ppacketHeaderRec->source.network,
          packetHeader.destination.network,4);
   movmem(ppacketHeaderRec->source.node,
          packetHeader.destination.node,6);
   movmem(&socketDest, packetHeader.destination.socket,2);
   movmem(ppacketHeaderRec->source.node,
          ecbEmission.immediateAddress,6);
   movmem(message, packetDataReply, DATA_LENGTH);
   ecbEmission.ESRAddress = NULL;
   ecbEmission.fragmentCount = 2;
   ecbEmission.socketNumber = socketSnd;
   ecbEmission.fragmentDescriptor[0].address = &packetHeader;
   ecbEmission.fragmentDescriptor[0].size = sizeof(IPXHeader);
   ecbEmission.fragmentDescriptor[1].address =
          (BYTE *) packetDataReply;
   ecbEmission.fragmentDescriptor[1].size = DATA_LENGTH;
   while (ecbEmission.inUseFlag != 0) IPXRelinquishControl();
   IPXSendPacket(&ecbEmission);
}

void RicAgrawRequest(void)
{
  int j;
  disable();
  localTS = highTS + 1;
  request = TRUE;
  replyCounter = noNodes - 1;
  for(j=0;j<noNodes;j++)
    if(connectionList[j] != connectionList[0]){
           memcpy(&bufferSnd[0],&localTS,2);
           memcpy(&bufferSnd[2],&connectionList[0],2);
           SendRequest(connectionList[j],bufferSnd);
    }
  enable();
}

void RicAgrawEnter(void)
{
   int ind1;

   do{
     disable();
     ind1 = replyCounter;
     enable();
   }while (!bioskey(1) && (ind1 != 0));
   if (ind1) {
           printf("Cancelled by user intervention!\n");
           exit(1);
   }
   inSection = TRUE;
}

void ReceiveRequest(char *message)
{
  memcpy(&timeStamp,&message[0],2);
  memcpy(&sender,&message[2],2);
  highTS = max(highTS,timeStamp);
  later = (request && ((timeStamp > localTS) ||
     ((timeStamp == localTS) && (sender>connectionList[0])))
     || inSection;
  if (later)
    laterTab[sender-1] = TRUE;
  else
    SendReply(&packetHeaderRec,"Reply");
}

void ReceiveReply(void)
{
  replyCounter--;
}
```


BUPT

B4. Implementarea algoritmului pentru obținerea excluderii reciproce în accesul la k resurse -varianta Windows

```
/* ***************************************************************
      Server care implementeaza algoritmul excluderii
            reciproce in accesul la K resurse
   -Suportul teoretic este prezentat in capitolul 4, la 4.3.2.3.

*************************************************************** */
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <process.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
# include "raywerr.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define BROADCAST_PORT  6562
#define REMOTE_PORT  6560
#define LOCAL_PORT  6561
#define MAX_HOST_NO 20
#define MAX_STRING 20
#define MAX_BUFFER 256
#define LEN 400
#define TRUE      1
#define FALSE     0
# define DEBUG TRUE
/* ***************************************************************
 * Functii si variabile interne serverului                       *
 ****************************************************************/
void CreateBroadcastChan(void);
SOCKET CreateRemoteChan(void);
SOCKET CreateLocalChan(void);
SOCKET CreateClientChan(void);
void FindAddress(void);
int NotInCS(void);
void Debug(const char *c,...);
struct sockaddr_in sad, cad,cla,sva,lCad;
char connTab[MAX_HOST_NO][MAX_HOST_NO];
char  HostName[MAX_BUFFER];
int processNo,myID;
int replyCounter[MAX_HOST_NO];
int  bsd;

void main(int argc, char *argv[])
{
 struct sockaddr_in sdr;          /* adresa Inet */
 fd_set rfds;                     /* read file descriptor set*/
 fd_set afds;                     /* active file descriptor set*/
 SOCKET rChan,lChan,cli;          /* identificatorii canalelor */
  int sval,lCadl;                 /* folosite  la recvfrom */
  int sender, ts;       /* folosite pentru receptia mesajelor */
  int LocalTS=0, HighTS = 0; /* retin marcile de timp */
  BOOL inAccess=FALSE;        /*flag pentru intrarea in SC */
  char sBf[LEN], rBf[LEN];        /* buffere */
 HANDLE hStdin,hStdout, enterChan; /*enterChan este semafor*/
WSADATA WSAData;
 DWORD result;
 BOOL later,reqAccess=FALSE;          /* flag-uri */
 char  temp[LEN],mes[MAX_STRING];     /* buffere */
 int laterTab[MAX_HOST_NO]; /* no. mes. Reply de asteptat */
 int objNo;                           /* no. de obiecte */
 int nfds, err,i,status;              /* var. de lucru */
/****************************************************************
faza de initializare: se prieia din linia de comanda identificatorul
host-ului local si numarul de procese si numarul de obiecte
Se creaza canalele.
Fiecare server trimite broadcast identificatorul sau si adresa sa
Se pun in asteptare de mesaje canalele remoteChan, localChan
**************************************************************** */
FreeConsole();
AllocConsole();
SetConsoleTitle(" Server Exclusive Access");
hStdin=GetStdHandle(STD_INPUT_HANDLE);
hStdout=GetStdHandle(STD_OUTPUT_HANDLE);
if ((status = WSAStartup(MAKEWORD(1,1), &WSAData)) == 0)
  Debug( "WSA ok \n");
else
  Panic(ERROR12);
if(argc!=4) {
   printf("Usage : [name] [identifier] [hosts] [objects]\n");
  exit(0);
}
for(i = 0;i<MAX_HOST_NO;i++)
     laterTab[i]=0;
 enterChan = CreateSemaphore(NULL, 0L, 1L, "LocalSem");
myID = atoi(argv[1]);
processNo = atoi(argv[2]);
objNo = atoi(argv[3]);
CreateBroadcastChan();
ReadConsole(hStdin,temp,10,&result,NULL);
FindAddress();
for(i = 0;i<processNo;i++)
   Debug(" Host %d  of address %s \n",i,connTab[i]);
getchar();
closesocket(bsd);
rChan = CreateRemoteChan();
lChan = CreateLocalChan();
cli = CreateClientChan();
FD_ZERO(&afds);
FD_SET(rChan,&afds);
FD_SET(lChan,&afds);

/* *************************************************************** ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
*************************************************************** *** */
while(TRUE){
 Debug("Waiting messages... \n");
 CopyMemory((char *)&rfds, (char *)&afds, sizeof(rfds));
 if(select(nfds,&rfds,(fd_set *)0,(fd_set *)0,(struct timeval *)0)<0){
      err = WSAGetLastError();
      Debug("Select error: %d\n",err);
 }
 if(FD_ISSET(rChan,&rfds)){   /* s-a primit un mesaj din retea */
   sval = sizeof(sva);
  err = recvfrom(rChan, rBf, LEN,0,(struct sockaddr*)&sva,&sval);
  if(err<0)
     Debug("Recvfrom  -remote- error\n");
  sscanf(rBf,"%2d%3d%s",&sender,&ts,mes);
  Debug("%s from %d ts=  %d...\n",mes,sender,ts);
  if(strcmp(mes,"REQUEST") == 0){ /* tratare REQUEST */
     HighTS = max(HighTS,ts);
     later = (inAccess) || (reqAccess && ((ts > LocalTS) ||
            ((ts == LocalTS) && (sender > myID))));
    if (later){  /* se amina transmiterea mes. REPLY  */
      laterTab[sender]++ ;
      Debug("laterTab = ");
      for(i=0;i<10;i++)
        Debug(" %d",laterTab[i]);
      Debug("\n");
    }
   else{  /* se poate transmite REPLY */
     memset((void FAR *)&sdr, 0,  sizeof(sdr));
     sdr.sin_family     = AF_INET ;
     sdr.sin_addr.s_addr = inet_addr(connTab[sender]) ;
     sdr.sin_port       = htons(REMOTE_PORT) ;
     sprintf(sBf,"%2d %3d %s",myID,1,"REPLY");
     Debug("Sending:%s\n",sBf);
     sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
   }
 }
 else
  if(strcmp(mes,"REPLY") == 0){ /* tratare REPLY */
   replyCounter[sender] = replyCounter[sender] - ts;
   if(reqAccess && (NotInCS() >= processNo - objNo)){
```


BUPT

```
        inAccess = TRUE;
        reqAccess = FALSE;
        ReleaseSemaphore(enterChan,1,NULL);
      }
    }
  }
if(FD_ISSET(lChan,&rfds)){ /* s-a primit un mesaj de la client */
 lCadl = sizeof(lCad);
err =recvfrom(lChan,rBf,LEN,0,(struct sockaddr*)&lCad,&lCadl);
if(err<0)
   Debug("Recvfrom -local-  error \n");
 sscanf(rBf,"%s",mes);
 Debug("Received from my cli %s.\n",mes);
 if(strcmp(mes,"REQUEST") == 0){ /* tratare REQUEST */
    reqAccess = TRUE;
    LocalTS = HighTS + 1;
  for(i=0;i<processNo;i++){
    if(i!= myID){
     memset((void FAR *)&sdr, 0,  sizeof(sdr));
     sdr.sin_family      = AF_INET ;
      sdr.sin_addr.s_addr = inet_addr(connTab[i]) ;
     sdr.sin_port         = htons(REMOTE_PORT) ;
     sprintf(sBf,"%2d %3d %s",myID,LocalTS,"REQUEST");
     Debug("Sending:%s\n",sBf);
     sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      replyCounter[i]++;
    }
  }
 }
 else
 if(strcmp(mes,"RELEASE") == 0){ /* tratare RELEASE */
   inAccess = FALSE;
   for(i=0;i<processNo;i++)
     if(laterTab[i] != 0){
       memset((void FAR *)&sdr, 0,  sizeof(sdr));
      sdr.sin_family      = AF_INET ;
       sdr.sin_addr.s_addr = inet_addr(connTab[i]) ;
      sdr.sin_port         = htons(REMOTE_PORT) ;
      sprintf(sBf,"%2d %3d %s",myID,laterTab[i],"REPLY");
      Debug("Sending:%s\n",sBf);
       sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      laterTab[i] = 0;
     }
    }
   }
  }
}
/* ***************************************************************** **
*                        Functii folosite de server                  *
******************************************************************* ***/
int NotInCS(void)      /* contorizeaza nr. de procese care nu  */
{                      /* sunt in sectiune critica   */
 int cnt,j;
 cnt = 0;
  for(j=0;j<processNo; j++)
    if((j != myID) && (replyCounter[j] == 0))
    cnt++;
 return cnt;
}
void CreateBroadcastChan(void)
{
  int i;
  memset((void FAR *)&sad, 0,  sizeof(sad)) ;
 sad.sin_family    = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
 sad.sin_port    = htons(BROADCAST_PORT) ;
 if((bsd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR1);
 i = 1 ;
 if(setsockopt(bsd, SOL_SOCKET, SO_BROADCAST, (char *)
                                          &i, sizeof(i)))
    Panic(ERROR2);
  memset((void FAR *)&cad, 0, sizeof(cad)) ;
 cad.sin_family    = AF_INET ;
 cad.sin_addr.s_addr = htonl(INADDR_ANY) ;
 cad.sin_port    = htons(BROADCAST_PORT) ;
  if(bind(bsd, (struct sockaddr *) &cad, sizeof(cad)) < 0)
     Panic(ERROR3);
 }
void FindAddress(void) /* determina adresele host-urilor */
{
 struct in_addr adr;
  struct sockaddr cadr;
 struct hostent *host;
  int cadrl;
 char code[MAX_STRING], string[MAX_STRING];
  int idRec,i ;
  char sBf[ LEN];
if (gethostname(HostName,256) != 0)
    Panic(ERROR4);
 if((host = gethostbyname(HostName)) == NULL)
    Panic(ERROR5);
 if(host != NULL)
   adr.s_addr = ((struct in_addr *) (host->h_addr_list[0]))->s_addr;
 if(strcpy(code,inet_ntoa(adr)) == NULL)
    Panic(ERROR6);
 strcpy(connTab[myID], code);
 sprintf(sBf,"%2d %s",myID,code);
 sendto(bsd,sBf,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
 for(i=0;i<processNo;i++){
    cadrl = sizeof(cad) ;
    recvfrom(bsd, sBf, LEN, 0, (struct sockaddr*)&cadr, &cadrl) ;
    sscanf(sBf,"%2d%s",&idRec,string);
    if(idRec != myID)
    strcpy(connTab[idRec],string);
  }
}
SOCKET CreateRemoteChan(void) {
 SOCKET s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
      Panic(ERROR7);
 memset((void FAR *)&sva, 0, sizeof(sva)) ;
 sva.sin_family      = AF_INET ;
 sva.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sva.sin_port        = htons(REMOTE_PORT) ;
 if(bind(s, (struct sockaddr *)&sva, sizeof(sva)) < 0)
   Panic(ERROR8);
 return s;
}
SOCKET CreateLocalChan(void) {
 SOCKET s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   Panic(ERROR9);
 memset((void FAR *)&lCad, 0, sizeof(lCad)) ;
 lCad.sin_family      = AF_INET ;
 lCad.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 lCad.sin_port        = htons(LOCAL_PORT) ;
 if(bind(s, (struct sockaddr *)&lCad, sizeof(lCad)) < 0)
  Panic(ERROR10);
 return s;
}
SOCKET CreateClientChan() {
  SOCKET s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   Panic(ERROR11);
 memset((void FAR *)&cla, 0, sizeof(cla)) ;
 cla.sin_family      = AF_INET ;
 cla.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 cla.sin_port        = htons(REMOTE_PORT) ;
 return s;
}
void Debug(const char *c,...){
  va_list args;
  if (DEBUG) {
  va_start(args,c);
  vprintf(c,args);
  va_end(args);
  }
 }
void Panic( int err){
  printf("Exit:  %d error \n", err);
  exit(0);
 }
```


BUPT

B5. Implementarea algoritmului bazat pe un token circulant pentru obținerea excluderii reciproce varianta Novell NetWare

```
/* *************************************************************
      Server care implementeaza  algoritmul  bazat pe
                    un token circulant
   -Suportul teoretic este prezentat in capitolul  4, la 4.3.3.1.
 * ************************************************************ */
#include <nxt.h>
#include <nit.h>
#include <stdio.h>
#include <dos.h>
#include <mem.h>
#include <string.h>
#include "exclacc.h"
/* ************************************************************ **
 *  Functii si variabile interne serverului                      *
 ************************************************************ ***/
#define DATA_LENGTH 80
int Initialize(void);     /* asigura initializare IPX */
int InitSend(void);       /* asigura  initializarea unui ECB pt. emisie*/
int InitReceive(void); /* asigura initializarea unui ECB pt. receptie*/
int GetConnectionNumbers
           (char users[MAX_CON][USER_LEN]);
void far _loadds ReceiveESR(void);  /* rutina ESR */
void SendToken(char *message);    /* trimite token-ul  la vecin */
void TokenRequest(void);  /* cere token-ul */
void TokenEnter(void);        /* blocare pina la primirea token-ului*/
void ReceiveToken(void);  /* apelata la primirea token-ului */
int noNodes;           /* numarul de statii */
int request;           /* cerere token */
int enter;             /* indica accesul in SC */
int myPosition, nextPosition;
int noOb;
WORD connectionList[30];  /* nr de ale conexiune statiilor */
IPXHeader nextAddress;     /* zone pt. header */
IPXHeader packetHeaderSnd;
IPXHeader packetHeaderRec;
char packetDataRec[DATA_LENGTH]; /* buffere */
char packetDataToken[DATA_LENGTH];
char bufferSnd[DATA_LENGTH];
ECB ecbEmission;
ECB ecbReception;
WORD socketRec = 0x5000;
WORD socketDest=0x5000;
WORD socketSnd=0x6000, socket;
int code;

int ExclAccessInitialize(void)  /* conectare la sistem */
{
  int val;
  if ((val = Initialize()) != 0) return val;
  if ((val = InitSend()) != 0) return val;
  if ((val = InitReceive()) != 0) return val;
  if (connectionList[myPosition] == connectionList[0])
     SendToken("OK");
  return 0;
}
int SetUsers(char users[MAX_CON][USER_LEN],
                     int hosts, int noObjects)
{ /* cunoscind numele user-ilor de pe fiecare statie, determina */
 /* numerele de conexiune  pentru toate statiile */
  int value;
  noNodes = hosts;
  value = GetConnectionNumbers(users);
  if (value != 0) return value;
  noOb = noObjects;
  return 0;
}
int Initialize(void)        /* initializare IPX */
{
 if ((code = IPXInitialize()) != 0)
   return 1;
 else
   return 0;
}
int InitSend(void)
{
if ((code = IPXOpenSocket((BYTE*)&socketSnd,0x00)) != 0)
   return 2;
else
   return 0;
}
int InitReceive(void)
{
 int code;
  /* initializarea structurilor de date */
 ecbReception.ESRAddress = ReceiveESR;
 ecbReception.fragmentCount = 2;
 ecbReception.socketNumber = socketRec;
 ecbReception.fragmentDescriptor[0].address =
                                        &packetHeaderRec;
 ecbReception.fragmentDescriptor[0].size = sizeof(IPXHeader);
 ecbReception.fragmentDescriptor[1].address = (BYTE *)
                                        packetDataRec;
 ecbReception.fragmentDescriptor[1].size = DATA_LENGTH;
 /* deschiderea soclului si intrarea in "ascultare" de pachete */
 if ((code = IPXOpenSocket((BYTE*)&socketRec,0x00)) != 0)
    return 3;
 IPXListenForPacket(&ecbReception);
 IPXRelinquishControl();
 return 0;
}
int GetConnectionNumbers(char users[MAX_CON][USER_LEN])
{
  int i=0;  int sort;
  WORD numberOfConnections = 0, saved;
  WORD maxConnections = MAX_CON;
  WORD myConnectionNo;
  BYTE serverNode[6];
  BYTE serverNetwork[4];
  int j;
  for(i=0;i<noNodes;i++)
  {
   do{
     GetObjectConnectionNumbers(users[i], OT_USER,
     &numberOfConnections,&connectionList[i], maxConnections);
     if(numberOfConnections == 0)
       return i;
     }while(numberOfConnections == 0);
   }
 myConnectionNo = connectionList[0];
 do{  /* ordoneaza tabela de conexiuni */
   sort=TRUE;
   for(i=0;i<noNodes-1;i++)
   if(connectionList[i] > connectionList[i+1]){
      saved = connectionList[i];
      connectionList[i]=connectionList[i+1];
      connectionList[i+1]=saved;
     sort=FALSE;
   }
  }while(!sort);
 for(j=0;j<noNodes;j++){
    if(connectionList[j] == myConnectionNo)
      myPosition = j;
  }
 if(myPosition == noNodes -1)
   nextPosition = 0;
 else
  nextPosition = myPosition+1;
 GetInternetAddress(connectionList[nextPosition],(char *)
           serverNetwork,(char *)serverNode,&socket);
 movmem(serverNetwork,
                     packetHeaderSnd.destination.network,4);
 movmem(serverNode, packetHeaderSnd.destination.node,6);
 movmem(serverNode, ecbEmission.immediateAddress,6);
 movmem(&socketDest, packetHeaderSnd.destination.socket,2);
 return 0;
}
void ExclAccessEnd(void)
```


BUPT

```
{
  IPXCancelEvent(&ecbReception);
  IPXCloseSocket(socketRec);
  IPXCloseSocket(socketSnd);
}
void ExclAccessOpen(void) /* pentru obtinerea accesului in SC */
{
  TokenRequest();
  TokenEnter();
}
void ExclAccessClose(void) /* pentru iesirea din SC */
{
 disable();
 request = FALSE;
 SendToken("OK");
 enter = FALSE;
 enable();
}
void far _loadds ReceiveESR(void)
{ /* apelata la sosirea unui pachet */
 disable();
 if (ecbReception.completionCode == 0){
   if(strcmp(packetDataRec, "OK") == 0)
      ReceiveToken();
   }
 enable();
 IPXListenForPacket(&ecbReception);
}
void SendToken (char *message)
{
   movmem(message, packetDataToken, DATA_LENGTH);
   ecbEmission.ESRAddress = NULL;
   ecbEmission.fragmentCount = 2;
   ecbEmission.socketNumber = socketSnd;
   ecbEmission.fragmentDescriptor[0].address =
                                  &packetHeaderSnd;
   ecbEmission.fragmentDescriptor[0].size = sizeof(IPXHeader);
   ecbEmission.fragmentDescriptor[1].address = (BYTE *)
                                  packetDataToken;
   ecbEmission.fragmentDescriptor[1].size = DATA_LENGTH;
   while (ecbEmission.inUseFlag != 0) IPXRelinquishControl();
   IPXSendPacket(&ecbEmission);
}

void TokenRequest(void)
{  request = TRUE;
}

void TokenEnter(void) /*  asteptare pina la intrarea in SC */
{
   int enter1;
   do{
     disable();
     enter1=enter;
     enable();
   }while(!enter1);
}

void ReceiveToken(void)  /*  apelata la primirea token-ului */
{
 int localRequest;
 localRequest = request;
 if(!localRequest)
    SendToken("OK");
 else
    enter = TRUE;
}
```

B6. Implementarea algoritmului Chandy pentru obținerea excluderii reciproce -varianta Unix

```
/* ************************************************************** *
     Server  care implementeaza  algoritmul  Chandy
   -Suportul teoretic este prezentat in capitolul  4, la 4.3.3.2.
************************************************************** */
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdarg.h>
#include "Chanduer.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define REMOTE_PORT  6560
#define LOCAL_PORT   6561
#define BROADCAST_PORT   6562
#define FIFO_NAME "enterChan"
#define PERMS 0666
#define MAX_HOST_NO      20
#define MAX_HOST_NAME    256
#define MAX_TYPE_MES_LEN 10
#define LEN   400
#define MAX_NO_PACKETS   100
#define MAX_STRING       20
#define REQUEST 1
#define TOKEN   0
#define TRUE            1
#define FALSE           0
#define DEBUG TRUE
/* ************************************************************** **
 *  Functii si variabile interne serverului                        *
************************************************************** ***/
void CreateBroadcastChan(void); /* creare canal broadcast */
int CreateRemoteChan(void);      /* creare  canal remoteChan*/
int CreateLocalChan(void);       /* creare  canal localChan */
int CreateClientChan(void);
void CreateEnterChan(void);      /* creare  canal enterChan*/;
void CloseEnterChan(void);
void FindAddresse(void);   /* determina adresele host-urilor*/
void Debug(const char *c, ...);

int enterChan ;     /* canal implementat prin FIFO */
int        bsd;          /*  soclu pt. transmisii broadcast */
struct sockaddr_in sad, cad,cla,sva,lCad;/* adrese Inet */
char connTab[MAX_HOST_NO][MAX_HOST_NO];
int processNo;   /* numarul de noduri participante la alg.*/
int myID;            /* identificatorul host-ului local */

void main(int argc, char *argv[] )
{
  struct sockaddr_in sdr;           /* adresa Inet */
  int rChan,lChan,cli;              /* identificatorii canalelor */
  fd_set rfds;                      /* read file descriptor set*/
  fd_set afds;                      /* active file descriptor set*/
  int sval,lCadl;                   /* folosite  la recvfrom */
  int inAccess = FALSE;             /* indica accesul in SC */
  int tokenOwner = FALSE;           /* indica posesia token-ului */
  int knd;                          /* tipul mesajelor */
  char sBf[LEN],sBf2[LEN]; /* buffere */
```


BUPT

```
char rBf[LEN],sBf3[LEN],mes[MAX_TYPE_MES_LEN];
int sender;              /* folosit pentru receptia mesajelor */
int reqQ[MAX_NO_PACKETS]; /* coada de cereri */
int  p =0;                /* nr. de cereri ale host-ului */
int T[MAX_HOST_NO];
int nfds, err, poz , i,j;  /* var de lucru */
char *pdest;

/* ************************************************************ ***
faza de initializare:  se prieia din linia de comanda identificatorul
host-ului local si numarul  de procese.
Se creaza canalele.
Fiecare server trimite broadcast identificatorul sau si adresa sa
Se pun in asteptare de mesaje canalele remoteChan, localChan
Hostul cu identificatorul cel mai mic primeste token-ul
************************************************************* *** */
 if(argc!=3) {
       printf("Usage : [name] [dentifier] [hosts] \n");
       exit(0);
 }
 for(i=0;i<MAX_NO_PACKETS;i++)
   reqQ[i] = 0;
 for(i=0;i<MAX_NO_PACKETS;i++)
   T[i] = 0;
 myID = atoi(argv[1]);
 processNo = atoi(argv[2]);
 CreateBroadcastChan();
 CreateEnterChan();
 getchar();
 FindAddresse();
 for(i = 0;i<processNo;i++)
     Debug("Host %d of adresse %s \n",i,connTab[i]);
  rChan = CreateRemoteChan();
 lChan = CreateLocalChan();
 cli = CreateClientChan();
 nfds = getdtablesize();
 FD_ZERO(&afds);
 FD_SET(rChan,&afds);
 FD_SET(lChan,&afds);
 if(myID == 0){
    tokenOwner = TRUE;
 }
/* ************************************************************ ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
************************************************************* *** */
while(1){
  Debug("Waiting messages...n");
  bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
  if(select(nfds,&rfds,(fd_set *)0,(fd_set *)0,(struct timeval *)0)<0)
     Debug("Select error %d \n",ermo);
  if(FD_ISSET(rChan,&rfds)){  /* mesaj din retea */
   sval = sizeof(sva);
   err=recvfrom(rChan,rBf,LEN,0,(struct sockaddr*)&sva, &sval) ;
   if(err<0)
      Debug("eroare la recvfrom \n");
   Debug("Received : %s \n",rBf);
   sscanf(rBf,"%2d-%2d",&knd,&sender);
   Debug("Message type : %d \n",knd);
   Debug("sender : %d \n",sender);
   switch(knd){
    case 0:
       strcpy(mes,"TOKEN");
       break;
    case 1:
       strcpy(mes,"REQUEST");
      break;
     default:
       Debug(" Unknown type\n");
       break;
   }
   Debug("Received from %d mes:= %s...\n",sender,mes);
   if(strcmp(mes,"REQUEST") == 0){ /* tratare REQUEST rem */
     j = 0;
     sscanf(rBf,"%2d-%2d-%2d",&knd,&sender,&reqQ[sender]);
     Debug("In reqQ[%d] = %d \n",sender,reqQ[sender]);
     if(tokenOwner && (!inAccess)){
       if(reqQ[myID] !=0){
         inAccess = TRUE;
        T[myID]=reqQ[myID];
        Debug("Push in T[%d]=%d",myID,reqQ[myID]);
        sprintf(sBf,"%s","enter");
         write(enterChan,sBf,6) ;   /* deblocheaza clientul local*/
       }
      else
      do{
       if(reqQ[j]!=0){                 /* paseaza token-ul */
         bzero((char *)&sdr, sizeof(sdr)) ;
        sdr.sin_family     = AF_INET ;
         sdr.sin_addr.s_addr = inet_addr(connTab[j]) ;
        sdr.sin_port       = htons(REMOTE_PORT) ;
         poz= sprintf(sBf,"%2d-%2d-", TOKEN,myID);
         for(i=0;i<processNo;i++)
            poz+=sprintf(sBf+poz,"%2d-",T[i]);
         Debug("Sending token with sBf =:%s\n",sBf);
         sendto(cli,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
       tokenOwner = FALSE;
      }
      j++;
     }while((j<processNo) && tokenOwner);
    }
   }
   else
    if(strcmp(mes,"TOKEN") == 0){ /* tratare TOKEN remote */
       Debug("ReceivedToken :%s ",rBf);
       pdest = strchr( rBf,'-');
        for(i=0;i<processNo;i++){      /* actualizeaza T */
         pdest = strchr( pdest+1,'-');
         sscanf(pdest+1,"%2d",&T[i]);
       }
       for(i=0;i<10;i++)
       Debug("  T[%d] = %d",i,T[i]);
       Debug("\n");
       T[myID] = reqQ[myID];
       inAccess = TRUE;
       tokenOwner = TRUE;
       sprintf(sBf,"%s","enter");
       write(enterChan,sBf,6) ;    /* deblocheaza clientul local*/
    }
  }
  if(FD_ISSET(lChan,&rfds)){   /* mesaj de la clientul local */
   lCadl = sizeof(lCad);
   err=recvfrom(lChan,rBf,LEN,0,(struct sockaddr*)&lCad,&lCadl);
   if(err<0)
     Debug("Recvfrom  -local- error \n");
   sscanf(rBf,"%s",mes);
   Debug("Received from a local client %s\n",mes);
   if(strcmp(mes,"REQUEST")==0){
    p++;                        /* contorizeaza o alta cerere */
    for(i=0;i<processNo;i++){   /* trimite cererea in retea */
    bzero((char *)&sdr, sizeof(sdr)) ;
    sdr.sin_family   = AF_INET ;
    sdr.sin_addr.s_addr = inet_addr(connTab[i]) ;
    sdr.sin_port  = htons(REMOTE_PORT) ;
    sprintf(sBf3,"%2d-%2d-%2d",REQUEST,myID,p);
    Debug("Sending REQ:%s la %d\n",&sBf3,i);
    sendto(cli,sBf3,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
   }
  }
  else
  if(strcmp(mes,"RELEASE") == 0){  /* tratare RELEASE local */
  inAccess = FALSE;
  reqQ[myID] = 0;
  j = 0;
  for(i=0;i<10;i++)
    Debug("  reqQ[%d] = %d",i,reqQ[i]);
  Debug("\n");
  for(i=0;i<10;i++)
    Debug("  T[%d] = %d",i,T[i]);
  Debug("\n");
  do{                            /* paseaza token-ul la alt host */
   if(reqQ[j]!=0)
    if(reqQ[j] > T[j]){
     bzero((char *)&sdr, sizeof(sdr)) ;
```


BUPT

```
      sdr.sin_family      = AF_INET ;
      sdr.sin_addr.s_addr = inet_addr(connTab[j]) ;
      sdr.sin_port        = htons(REMOTE_PORT) ;
      poz = sprintf(sBf2,"%2d-%2d-",TOKEN,myID);
      for(i=0;i<processNo;i++)
         poz+=sprintf(sBf2+poz,"%2d-",T[i]);
      Debug("Sending TOKEN for RELEASE:%s\n",sBf2);
      sendto(cli,sBf2,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      tokenOwner = FALSE;
    }
    else
     reqQ[j] = 0;
     j++;
  }while((j<processNo) && tokenOwner);
  Debug("After send the reqQ is: ");
  for(i=0;i<10;i++)
    Debug("  reqQ[%d] = %d",i,reqQ[i]);
  Debug("\n");
  }
 }
 }
}
/* ************************************************************** **
 *                    Functii folosite de server                   *
 ************************************************************** ***/
void CreateBroadcastChan(void)
/* creaza canalul pentru transmisii broadcast  */
{
 int i;
 bzero((char *)&sad, sizeof(sad)) ;
 sad.sin_family      = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
 sad.sin_port        = htons(BROADCAST_PORT) ;
 if((bsd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   Panic(ERROR1);
 i = 1 ;

if(setsockopt(bsd,SOL_SOCKET,SO_BROADCAST,(char*)&i,sizeof(i)))
   Panic(ERROR2);
bzero((char *)&cad, sizeof(cad)) ;
 cad.sin_family      = AF_INET ;
 cad.sin_addr.s_addr = htonl(INADDR_ANY) ;
 cad.sin_port        = htons(BROADCAST_PORT) ;
 if(bind(bsd, (struct sockaddr *) &cad, sizeof(cad)) < 0)
   Panic(ERROR3);
}
void FindAddresse(void)
/* memoreaza in tabela de adrese adresele si identificatorii */
/* tuturor host-urilor                                       */
{
 struct in_addr adr;
 struct sockaddr cadr;
 struct hostent *host;
 int cadrl;
 char sBf[LEN];
 char HostName[MAX_HOST_NAME];
 char code[MAX_STRING]; char string[MAX_STRING];
 int idRec,i;
 if (gethostname(HostName,256) != 0)
        Panic(ERROR4);
 if((host = gethostbyname(HostName)) == NULL)
       Panic(ERROR5);
 if(host != NULL)
   adr.s_addr = ((struct in_addr *) (host->h_addr_list[0]))->s_addr;
 if(strcpy(code,inet_ntoa(adr)) == NULL)
      Panic(ERROR6);
 strcpy(connTab[myID], code);
 sprintf(sBf,"%2d %s",myID,code);
 sendto(bsd,sBf,LEN,0,(struct sockaddr*)&sad,sizeof(sad));
 for(i=0;i<processNo;i++){
    cadrl = sizeof(cad) ;
    recvfrom(bsd, sBf, LEN,0,(struct sockaddr*)&cadr, &cadrl) ;
    sscanf(sBf,"%2d%s",&idRec,string);
    if(idRec != myID)
       strcpy(connTab[idRec],string);
 }
}

int CreateRemoteChan(void)
/* creaza canalul pentru transmisii in retea  */
{
 int s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   Panic(ERROR7);
 bzero((char *)&sva, sizeof(sva)) ;
 sva.sin_family      = AF_INET ;
 sva.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sva.sin_port        = htons(REMOTE_PORT) ;
 if(bind(s, (struct sockaddr *)&sva, sizeof(sva)) < 0)
   Panic(ERROR8);
 return s;
}

int CreateLocalChan(void)
/* creaza canalul pentru receptia mesajelor de  la clientul local */
{
 int s;

 if((s = socket(AF_INET,.SOCK_DGRAM, 0)) < 0)
   Panic(ERROR9);
 bzero((char *)&lCad, sizeof(lCad)) ;
 lCad.sin_family      = AF_INET ;
 lCad.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 lCad.sin_port        = htons(LOCAL_PORT) ;
 if(bind(s, (struct sockaddr *)&lCad, sizeof(lCad)) < 0)
   Panic(ERROR10);
 return s;
}

int CreateClientChan()
/* creaza canalul pentru emisia mesajelor  in retea */
{
  int s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   Panic(ERROR11);
 }
 bzero((char *)&cla, sizeof(cla)) ;
 cla.sin_family      = AF_INET ;
 cla.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 cla.sin_port        = htons(REMOTE_PORT) ;
 return s;
}
void CreateEnterChan(void)
/* creaza canalul pentru deblocarea clientului local */
{
 unlink(FIFO_NAME);
 if( mknod(FIFO_NAME, S_IFIFO I PERMS , 0)<0)
   Panic(ERROR12);

 if((enterChan = open(FIFO_NAME, O_RDWR))<0)  {
   Panic(ERROR13);
}
void CloseEnterChan(void)
{
 unlink(FIFO_NAME);
}
void Debug(const char *c, ...){
  va_list args;
 if(DEBUG){
  va_start(args,c);
  vprintf(c, args);
  va_end(args);
  }
 }
void Panic( int err)
{
 printf("Exit:  %d error \n", err);
 exit(0);
}
```


BUPT

B7. Implementarea semafoarelor distribuite -varianta Windows

```
/* ****************************************************************
        Server   pentru  semafoare distribuite
-Serverul este utilizat pentru controlul accesului la o baza de
 date intr-o retea locala  folosind semafoare distribuite
 (anexa C4)
 -Suportul teoretic este prezentat in capitolul 4, la 4.3.4.1
***************************************************************** */
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <string.h>
#include <process.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
# include "semwerr.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define BROADCAST_PORT  6560
#define LOCAL_PORT  6561
#define MAX_SEM    16         /* nr. maxim de semafoare */
#define MAX_MES_QUEUE  100 /* lung. max. a cozii de mes. */
#define MAX_ACK_QLEN   1000  /* lung. max. a cozii de ack. */
#define LEN 100       /* lungimea max.ima  a unui  mesaj */
#define TRUE       1
#define FALSE      0
#define DEBUG      1

/* ************************************************************** **
 * Functii si variabile interne serverului                        *
 ************************************************************** ***/
typedef enum { UP, DOWN, ACK } kind;  /* actiuni asupra sem.*/
typedef struct {/* structura unui element din coada de op. necon.*/
  kind k ;          /* tipul mesajului */
  int  sender ;     /* identificatorul emitatorului */
  int  ts ;         /* marca de timp */
  int sem ;         /*  semaforul */
} QUEUE ;
typedef struct  {/* structura unui element din coada de Ack*/
 int sender;       /* identificatorul emitatorului */
 int ts ;           /* marca de timp */
 int sem ;         /*  semaforul */
} ACKNOWLEDGE ;

SOCKET CreateRemoteChan(void);
SOCKET CreateLocalChan(void);
void CreateAnswerChan(void);
void CloseAnswerChan(void);
void InsertQueue(int sender , kind k , int ts, int sem) ;
void RemoveQueue(int poz) ;
void InsertAck(int sender, int ts, int sem) ;
int    FindFullyAck(kind k, int sem) ;
void Debug(const char *c, ...);

HANDLE answerChan ;      /* canalul answerChan */
QUEUE   mesQueue[MAX_MES_QUEUE] ; /* coada de op */
ACKNOWLEDGE  ackList[MAX_ACK_QLEN] ;
int  nUp[MAX_SEM], nDown[MAX_SEM];
int  maxHosts ;         /* numarul participantilor la algoritm */
int   maxQueue, maxAck;    /* dim. curenta a cozilor*/
LPSTR lpAnswer;
int   ifa ;
struct sockaddr_in sar,sal;  /* adrese pentru lucrul cu socket */

void main(int argc, char* argv[])
{
 SOCKET rChan,lChan;           /* canalel pentru comunicatii */
 struct sockaddr_in sdr;
 int sarl,sall;
 HANDLE hStdout;
 HANDLE  hAnswerChan ;     /* answerChan se implementeaza
             /* cu un semafor si o zona de memorie partajata  */
 char sBf[LEN],rBf[LEN];            /* buffere */
 fd_set rfds;                        /* read file descriptor set*/
 fd_set afds;                        /* active file descriptor set*/
 int faUp=0, faDown=0;               /* no. mesajelor  fully ack. */
 int  myid;                          /* identificatorul host-ului */
 int  localTS=0;                     /* marca de timp locala */
 WSADATA WSAData;
 int status;
 int Wait = FALSE;                   /* folosit la deblocare clientului */
 int SemWait = -1;                   /* folosit la deblocare clientului */
 int nfds, err;                      /* variabile de lucru */
 int nr, sem;
 int  ts, sender; char ck;           /* componente in mesaje */
 kind k;
 int i;


/* ************************************************************** ***
faza de initializare:  se prieia din linia de comanda identificatorul
host-ului local si numarul  de procese.
Se creaza canalele.
Se pun in asteptare de mesaje canalele remoteChan, localChan
*************************************************************** *** */

FreeConsole();
AllocConsole();
SetConsoleTitle("Win32 - Server");
hStdout=GetStdHandle(STD_OUTPUT_HANDLE);
if ((status = WSAStartup(MAKEWORD(1,1), &WSAData)) == 0)
     Panic(ERROR4);
else
  Debug( "WSA not OK\n");
if(argc!=3) {
   printf("Usage : [name] [identifier] [hosts]\n");
   exit(0);
 }
myid=atoi(argv[1]);
maxHosts=atoi(argv[2]);
hAnswerChan = CreateSemaphore(NULL, 0L, 1L, "semhelp");
 localTS = 0 ;
 for(i=0;i<MAX_SEM; i++)
    nDown[i]= 0 ; nUp[i] = 0;
 CreateAnswerChan();
 rChan = CreateRemoteChan();
 lChan = CreateLocalChan();
 FD_ZERO(&afds);
 FD_SET(rChan,&afds);
 FD_SET(lChan,&afds);

/* ************************************************************** ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
*************************************************************** *** */
 while(TRUE){
  Debug("Waiting messages...\n");
  CopyMemory((char *)&rfds, (char *)&afds, sizeof(rfds));
  if(select(nfds,&rfds,(fd_set *)0,(fd_set *)0,(struct timeval*)0)<0){
     err = WSAGetLastError();
     Debug("Select error : %d",err);
  }

  if(FD_ISSET(rChan,&rfds)){  /* s-a primit un mesaj remote */
    sarl = sizeof(sar);
    err=recvfrom(rChan, rBf,LEN,0,(struct sockaddr*)&sar, &sarl) ;
   if(err<0)
      Debug("Recvfrom -remote- error \n");
   sscanf(rBf,"%2d-%c-%2d-%10d",&sender,&ck,&sem,&ts) ;
   switch(ck) {
    case 'V': k = UP; break;
    case 'P': k = DOWN; break;
    case 'A': k = ACK; break;
   }
   Debug("%c for %d from %d host ts=%d...\n",ck,sem,sender,ts);
   localTS = max(localTS, ts+1) ; /* actualizare TS */
   localTS++ ;
   if((sender == myid)&&(k == UP)){ /* se memoreaza deblocare*/
```


BUPT

```
     SemWait = sem;
     Wait = TRUE;
    }
    if((k == DOWN) || (k == UP)) { /* s-a primit DOWN sau UP */
      InsertQueue(sender, k, ts, sem);/* inserare mesaj in coada */
      memset((void FAR *)&sdr, 0,  sizeof(sdr));
      sdr.sin_family     = AF_INET ; /* si trimite confirmare */
      sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
      sdr.sin_port       = htons(BROADCAST_PORT) ;
      sprintf(sBf,"%2d-%c-%2d-%10d",myid,'A',sem,localTS);
      Debug("Sending ACK for %d sem %d host ...\n",sem,myid);
      Debug("Sent message %s\n",sBf);
      sendto(rChan,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
      localTS++;
    }
    else
    if(k == ACK) {  /* s-a primit confirmare de operatie */
      InsertAck(sender, ts, sem);  /* inserea za confirmarea */
     ifa=0;
    do {        /* cauta daca operatia  UP devine complet confirmata */
       faUp = FindFullyAck( UP , sem) ;
       if(faUp != -1) {
         RemoveQueue(faUp) ;
         nUp[sem]++ ;
         if(Wait && (SemWait == sem)){
           Wait =FALSE;
            nr = nUp[sem]-nDown[sem];
            sprintf(lpAnswer,"%d",nr);
             ReleaseSemaphore(hAnswerChan,1,NULL);
         }
       }
     }while(faUp != -1) ;
     Debug("Value  nUp=%d for %d sem.\n",nUp[sem],sem);
     ifa = 0;
    do { /* cauta daca operatia DOWN devine complet confirmata */
       faDown = FindFullyAck( DOWN, sem ) ;
       Debug(" nUp=%d ; nDown=%d\n", nUp[sem],nDown[sem]);
       if(( faDown != -1) && (nUp[sem] > nDown[sem])) {
         ifa--;
          sender=mesQueue[faDown].sender ;
          RemoveQueue( faDown) ;
          nDown[sem]++ ;
          if(sender == myid) {
            nr = nUp[sem]-nDown[sem];
           sprintf(lpAnswer,"%d",nr) ;            /* Deblochez la semul
local sem */
            Debug("Unblocking my client who's waiting at DOWN \n");
            ReleaseSemaphore(hAnswerChan, 1, NULL);
         }
       }
     }while(faDown != -1) ;
     Debug("Value  nDown=%d for %d sem.\n ",nDown[sem],sem);
    }
   }

   if(FD_ISSET(lChan,&rfds)){ /* s-a primit un mesaj local */
    sall = sizeof(sal);
    err = recvfrom(lChan, rBf, LEN,0, (struct sockaddr*)&sal, &sall);
    if(err<0){
      Debug("Recvfrom  -local- error\n");
    }
    sscanf(rBf,"%2d-%c",&sem,&ck);
    switch(ck) {
     case 'V': k = UP; break;
      case 'P': k = DOWN; break;
    }
    Debug("Received from client %c type for %d sem \n",ck,sem);
    if((k== DOWN)||(k==UP)){ /* trimite mes. in retea */
       memset((void FAR *)&sdr, 0,  sizeof(sdr));
       sdr.sin_family  = AF_INET ;
       sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
       sdr.sin_port  = htons(BROADCAST_PORT) ;
      if(k==DOWN)
         sprintf(sBf,"%2d-%c-%2d-%10d",myid,'P',sem,localTS);
      else
         sprintf(sBf,"%2d-%c-%2d-%10d",myid,'V',sem,localTS);
     Debug("Sent message:%s\n",sBf);
```

```
        sendto(rChan,sBf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
        localTS++;
      }
    }
  }
  CloseAnswerChan();
 }

/* ************************************************************** **
 *                       Functii folosite de server                  *
 ************************************************************** ***/

void InsertQueue(int sender, kind k, int ts, int sem)
/* InsertQueue - insereaza un element in coada  de op. */
/* astfel incat coada sa ramana ordonata */
{
 int i,j;
 if((k==UP) || (k==DOWN)) {
  if(maxQueue == 0) {
   mesQueue[0].sender = sender ;
   mesQueue[0].k = k ;
   mesQueue[0].ts = ts ;
   mesQueue[0].sem = sem ;
   maxQueue = 1 ;
  }
  else {
   i = 0 ;
   while((i < maxQueue) && (ts >= mesQueue[i].ts))
              i++ ;
   for(j = maxQueue-1; j >= i; j--)
    mesQueue[j+1] = mesQueue[j];
   maxQueue++ ;
   mesQueue[i].sender = sender ;
   mesQueue[i].k = k ;
   mesQueue[i].ts = ts ;
   mesQueue[i].sem = sem ;
  }
 }
}

void InsertAck(int sender, int ts, int sem)
/* insereaza un element in coada de confirmari */
{

 ackList[maxAck].sender = sender ;
 ackList[maxAck].ts = ts ;
 ackList[maxAck].sem = sem ;
 maxAck++ ;
}

int FindFullyAck(kind k, int sem)
/* FindFullyAck - returneaza operatiile  de tipul  complet */
 /* confirmate  din coada  de operatii */
{
 int i, j, t ;
 for(i=ifa; i<maxQueue; i++) {
  if((mesQueue[i].sem==sem) && (mesQueue[i].k==k)) {
   for(j=0;j<maxHosts;j++) {
    for(t=0;t<maxAck;t++)
    if( (ackList[t].sender == j) &&(ackList[t].ts>mesQueue[i].ts) &&
       (ackList[t].sem == sem ))
      break;
    if(t==maxAck)
     break;
   }
   if(j==maxHosts) {
    if(k==UP) ifa=0;
    else     ifa=i+1;
    return i;
   }
  }
 }
 return -1;
}

void RemoveQueue(int poz)
/* RemoveQueue - sterge elementul din pozitia poz din coada */
```


BUPT

```
{
 int i;
 for(i = poz; i < maxQueue; i++)
        mesQueue[i] = mesQueue[i+1] ;
 maxQueue-- ;
}
SOCKET CreateRemoteChan(void)
/* creaza canalul pentru transmisii broadcast */
 {
 SOCKET s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR1);
 memset((void FAR *)&sar, 0, sizeof(sar)) ;
 sar.sin_family      = AF_INET ;
 sar.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sar.sin_port        = htons(BROADCAST_PORT) ;
 if(bind(s, (struct sockaddr *)&sar, sizeof(sar)) < 0)
    Panic(ERROR2);
 return s;
}

SOCKET CreateLocalChan(void)
/* creaza canalul pentru receptia mesajelor de la clientul local */
 {
 SOCKET s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR3);
 memset((void FAR *)&sal, 0, sizeof(sal)) ;
 sal.sin_family      = AF_INET ;
 sal.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sal.sin_port        = htons(LOCAL_PORT) ;
 if(bind(s, (struct sockaddr *)&sal, sizeof(sal)) < 0)
    Panic(ERROR4);
 return s;
}

void CreateAnswerChan(void)
/* creaza zona de memorie partajata prin care se */
/* comunica cu clientul */
{
 answerChan = CreateFileMapping((HANDLE)0xFFFFFFFF,
  NULL, PAGE_READWRITE, 0, 0x1000, "SEMDEMO");
 if(answerChan == NULL)
    Panic(ERROR5);
 lpAnswer = (LPSTR)MapViewOfFile(answerChan,
FILE_MAP_WRITE, 0, 0, 0);
 if(lpAnswer == NULL)
    Panic(ERROR6);
}

void CloseAnswerChan(void)
{
 UnmapViewOfFile(lpAnswer);
}

void Debug(const char *c, ...){
        va_list args;
        va_start(args,c);
        vprintf(c,args);
        va_end(args);
}

void Panic( int err){
 printf("Exit:  %d error \n", err);
 exit(0);
}
```

```
/* ****************************************************************
      Functiile puse la dispozitia unui client
 care folosește semafoarele distribuite - varianta  Windows

 -Suportul teoretic este prezentat in capitolul 4, la 4.3.4.1
 -Functiile care pot fi folosite de un client  sunt:
  int SemaphoreInitialize(void)
  int SemaphoreUP(int sid)
   int SemaphoreDown(int sid)
  void SemaphoreEnd(void)

  - Comunicarea cu serverul  se face printr-o zona de memorie
 partajata iar sincronizarea  server - client se realizeaza  cu
 ajutorul unui  semafor

*****************************************************************/
#include <winsock.h>
#include <windows.h>
#include "resource.h"
#include "semaforw.h"

int OpenLocalChan(void);
int CreateLocalChan(void);
int GetServerAddress(void);

LPSTR lpAnswer;
int    lChan ;
struct sockaddr_in sad;

int SemaphoreDown(int sid)
/* Semaphore_Down- operatia DOWN asupra sem.sid */
{
 char info[LEN];
 int nr;
 if ( sid < 0 || sid >MAX_SEM-1)
     return -1;
 sprintf(info,"%2d-%c",sid, 'P');
 sendto(lChan,info, LEN, 0,
           (struct sockaddr *)&sad, sizeof(sad));

 /* blocare pina la  primirea raspunsului de la server */
 WaitForSingleObject(hAnswerChan,INFINITE) ;

 /* urmeaza citirea din zona de memorie  partajata   */
 /*                   a valorii semaforului            */
 sscanf(lpAnswer,"%d",&nr) ;
 return nr;
}

int SemaphoreUP(int sid)
/* SemaphoreUp- operatiaUP asupra sem. sid */
{
 char info[LEN];
 int nr;
 if ( sid < 0 || sid >MAX_SEM-1)
     return -1;
 sprintf(info,"%2d-%c",sid, 'V') ;
 sendto(lChan,info, LEN, 0,
                      (struct sockaddr *) &sad,sizeof(sad)) ;
  /* blocare pina la  primirea raspunsului de la server */
 WaitForSingleObject(hAnswerChan,INFINITE) ;
 sscanf(lpAnswer,"%d",&nr);
 return nr;
}

int SemaphoreInitialize(void)
/* pentru conectare la  server */
{
 int errCode;
   if ((errCode = CreateLocalChan()) != 0 )
     return errCode;
  if ((errCode = OpenLocalChan() ) !=  0)
    return errCode;
 return 0;
}

 char * GetMyAddress(void)
```


BUPT

```
{
 int errCode;
 if  ((errCode = GetServerAddress()) != 0)
    return NULL;
 else
   return MyServerAddress;
}


int GetServerAddress(void)
/*determina adresa serverului propriu*/
{
struct in_addr adr;
char HostName[MAX_HOST_NAME];
struct hostent *host;
 if (gethostname(HostName,MAX_HOST_NAME) != 0)
    return ERROR2;
 if((host = gethostbyname(HostName)) == NULL)
    return ERROR3;
 if(host != NULL)
    adr.s_addr = ((struct in_addr *)(host->h_addr_list[0]))->s_addr;
 if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
    return ERROR4;
 return 0;
}

int OpenLocalChan(void)
/* deschide semaforul si soclul pentru comunicare
   cu serverul ; daca nu reuseste, intoarce un cod de eroare
*/
 {
  int errCode;      /*cod eroare */
  hAnswerChan = OpenSemaphore(SYNCHRONIZE,TRUE,
"semhelp");
  if  (hAnswerChan == NULL)
     return ERROR1;
  if ((errCode=GetServerAddress() ) != 0)
     return errCode;
  memset((void FAR *)&sad, 0, sizeof(sad)) ;
  sad.sin_family      = AF_INET ;
  sad.sin_addr.s_addr = inet_addr(MyServerAddress) ;
  sad.sin_port        = htons(LOCAL_PORT) ;
  if((lChan = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     return ERROR5;
   return 0;
 }

int CreateLocalChan(void)
/* obtine accesul la zona de memorie partajata */
{
 HANDLE hmmf;  /* handler pentru mem. partajata */
 hmmf = CreateFileMapping((HANDLE)0xFFFFFFFF,
   NULL, PAGE_READWRITE, 0, 0x2000, "SEMDEMO");
 if(hmmf == NULL)
     return ERROR6;
 lpAnswer = (LPSTR)MapViewOfFile(hmmf, FILE_MAP_WRITE,
0, 0, 0);
 if(lpAnswer == NULL)
     return ERROR7;
 return 0;
 }

 void SemaphoreEnd(void)
 {
  closesocket(lChan);
  UnmapViewOfFile(lpAnswer);
 }
```

B8. Implementarea contorilor de eveniment distribuiţi -varianta UNIX

```
/* *************************************************************
          Server   pentru  contori de eveniment distribuiti
  -Serverul  este folosit in anexa B10 pentru   implementarea
excluderii reciproce cu ajutorul contorilor de evenimente distribuiti
si  a secventiatorilor distribuiti
- Suportul teoretic este prezentat in capitolul 4, la 4.3.4.2-
************************************************************ */
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdarg.h>
# include  "countuer.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define BROADCAST_PORT  6560
#define LOCAL_PORT  6561
#define FIFO_NAME  "answerChan"
#define PERMS 0666
#define MAX_COUNTER          10     /* nr. max de contori */
# define MAX_MES_QUEUE 100  /* dim max a cozii de mesaje */
#define MAX_ACK_QLEN  1000  /* dim max a listei de confir.*/
#define ACKN      4
#define ADVANCE   1
#define LEN     400
#define TRUE               1
#define FALSE              0
#define DEBUG     TRUE

/* ************************************************************* **
 *  Functii si variabile interne serverului                     *

************************************************************* ***/

typedef enum { ADV, READ, AW, ACK } kind ;
typedef struct {/* structura unui element din coada de op. necon.*/
    int sender;         /* identificatorul emitatorului */
    int ts;             /* marca de timp */
    int counter;        /*  semaforul */
}QUEUE;
typedef struct {        /* structura unui element din coada de Ack*/
    int sender;         /* identificatorul emitatorului */
    int ts;             /* marca de timp */
    int counter;        /* contorul */
}ACKNOWLEDGE;

int CreateRemoteChan(void);
int CreateLocalChan(void);
void CreateAnswerChan(void);
void CloseAnswerChan(void);
void InsertQueue(int sender, int ts, int counter);
void RemoveQueue(int poz);
void InsertAck(int sender, int ts, int counter);
int FindFullyAck(int sender, int counter);
void Debug(const char *c, ...);

int answerChan ;      /* handler pipe */
QUEUE mesQueue[MAX_MES_QUEUE];
ACKNOWLEDGE ackList[MAX_ACK_QLEN];
int maxQueue, maxAck;
int maxHosts;
struct sockaddr_in  sar,sal;
```


BUPT

```
void main( int argc, char *argv[])
{
 struct sockaddr_in sdr;
 int sarl,sall;
 int rChan,lChan;
 fd_set rfds;                    /* read file descriptor set*/
 fd_set afds;                    /* active file descriptor set*/
 int nE[MAX_COUNTER], nEVal[MAX_COUNTER];
 char sBuf[LEN],rBuf[LEN];       /* buffere */
 int localTs=0;                  /* marca de timp locala */
  int myID;                      /* identificatorul host-ului */
 int sender, ts;                 /* componente in mesaje */
 int poz;                        /* variabile de lucru */
 int cr, counter;
 int val=0;
 int i,  nfds, err;
 kind k;
 int kd;

/* ******************************************************************** ***
faza de initializare:  se prieia din linia de comanda identificatorul
host-ului local si numarul  de procese.
Se creaza canalele.
Se pun in asteptare de mesaje canalele remoteChan, localChan
******************************************************************** *** */
 if(argc!=3) {
       Debug("Usage : [nume] [nr.identif.] [nr.total de procese] \n");
       exit(0);
 }
 for(i=0;i<MAX_COUNTER;i++)
            nEVal[i]=-1;
 CreateAnswerChan();
 myID = atoi(argv[1]);
 maxHosts = atoi(argv[2]);
  rChan = CreateRemoteChan();
 lChan = CreateLocalChan();
 nfds = getdtablesize();
 FD_ZERO(&afds);
 FD_SET(rChan,&afds);
 FD_SET(lChan,&afds);

/* ******************************************************************** ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
******************************************************************** *** */
while(TRUE){
  Debug("Waiting messages...\n");
  bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
  if(select(nfds,&rfds,(fd_set *)0,(fd_set *)0, (struct timeval *)0)<0)
     Debug("Select error %d \n",errno);
  if(FD_ISSET(rChan,&rfds)){  /* mesaj din retea */
   sarl = sizeof(sar);
    err=recvfrom(rChan,rBuf,LEN,0,(struct sockaddr*)&sar,&sarl);
   if(err<=0){
      Debug("Recvfrom -remote- error%d\n",err);
  }
  sscanf(rBuf,"%2d-%2d-%6d-%2d",&kd,&sender,&ts,&cr);
  Debug("From %d  %d ts = %d counter %d...\n",sender,kd,ts,cr);
  switch(kd){
    case 1: k = ADV;
            break;
    case 4: k = ACK;
             break;
  }
  localTs=max(localTs,ts+1);    /* actualizare marca de timp */
  localTs++;
  if(k==ADV){          /* tratare advance */
    InsertQueue(sender,ts,cr); /* insereaza mesajul in coada */
    bzero((char *)&sdr, sizeof(sdr)) ;
    sdr.sin_family  = AF_INET ;
    sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
    sdr.sin_port   = htons(BROADCAST_PORT) ;
    sprintf(sBuf,"%2d-%2d-%6d-%2d",ACKN,myID,localTs,cr);
    Debug("Sending:%s\n",sBuf);  /* trimite confirmarea */
    sendto(rChan,sBuf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
     localTs++;
   }
   else
     if(k==ACK){  /* tratare ACK */
     InsertAck(sender,ts,cr);
     for(i=0;i<maxAck;i++)
          Debug("ackList = %d ",ackList[i].ts);
     Debug("\n");
     for(i=0;i<maxQueue;i++)
          Debug("mesQueue = %d ",mesQueue[i].ts);
    Debug("\n");
    poz = FindFullyAck(sender, cr);
    if(poz>=0){
      counter = mesQueue[poz].counter;
      nE[counter]++;
       RemoveQueue(poz);/*poz = pozitia celui scos din coada*/
      Debug("nE[%d]= %d \n",counter,nE[counter]);
      if((nE[counter]>= nEVal[counter]) && (nEVal[counter] != -1)){
         Debug("nEVal = %d\n",nEVal[counter]);
         nEVal[counter] = -1;
        sprintf(sBuf,"%s","gata");
        write(answerChan,sBuf,6) ;
     }
    }
  }
 }
 if(FD_ISSET(lChan,&rfds)){    /* mesaj de la clientul local */
  sall = sizeof(sal);
   err=recvfrom(lChan,rBuf,LEN, 0, (struct sockaddr*)&sal, &sall);
   if(err<=0)
      Debug("Recvfrom - local -error\n");
   sscanf(rBuf,"%2d",&kd);
   switch(kd){
    case 2: k=AW;
     break;
    case 3: k = READ;
     break;
   case 1:k=ADV;
    break;
   }
  if(k==ADV){          /* tratare advanve */
    sscanf(rBuf,"%2d-%2d",&kd,&cr);
    bzero((char *)&sdr, sizeof(sdr)) ;
    sdr.sin_family      = AF_INET ;
    sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
    sdr.sin_port        = htons(BROADCAST_PORT) ;

sprintf(sBuf,"%2d-%2d-%6d-%2d",ADVANCE,myID,localTs,cr);
    Debug("Sending:%s\n",sBuf);
    sendto(rChan,sBuf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
    localTs++;
  }
  if(k==READ){        /* tratare read */
   sscanf(rBuf,"%2d-%2d",&kd,&cr);
   Debug("Received  %d and counter = %d...\n",kd,cr);
   Debug("nE[%d] pe READ local este : %d \n",cr,nE[cr]);
   sprintf(sBuf,"%d",nE[cr]);
   write(answerChan,sBuf,6) ;
  }
  else
   if(k==AW){  /* tratare await */
     sscanf(rBuf,"%2d-%2d-%2d",&kd,&cr,&val);
     Debug(" %d with counter = %d and val=%d...\n",kd,cr,val);
     nEVal[cr] = val;
     Debug("nEVal=%d\n",nEVal[cr]);
     if(nE[cr] >= nEVal[cr]){
        nEVal[cr] = -1;
        sprintf(sBuf,"%s","gata");
        write(answerChan,sBuf,6) ;
     }
    }
  }
 }
}
/* ************************************************************** **
 *                   Functii folosite de server                   *
 ************************************************************** ***/
```


BUPT

```
void InsertQueue(int sender, int ts, int counter)
/* insereaza un element in coada */
/* astfel incat coada sa ramana ordonata */
{
 int i,j;
 if(maxQueue == 0) {
  mesQueue[0].sender = sender ;
  mesQueue[0].ts = ts ;
  mesQueue[0].counter = counter ;
  maxQueue = 1 ;
 }
 else {
  i = 0 ;
  while((i < maxQueue) && (ts >= mesQueue[i].ts))
          i++ ;
  for(j = maxQueue-1; j >= i; j--)
   mesQueue[j+1] = mesQueue[j];
  maxQueue++ ;
  mesQueue[i].sender = sender ;
  mesQueue[i].ts = ts ;
  mesQueue[i].counter = counter ;
 }
}

void InsertAck(int sender, int ts, int counter)
/* insereaza in coada de Ack o confirmare */
{
  ackList[maxAck].sender = sender ;
  ackList[maxAck].ts = ts ;
  ackList[maxAck].counter = counter ;
  maxAck++;
}

int FindFullyAck(int sender, int counter)
/* FindFullyAck - returneaza operatiile de tipul complet */
 /* confirmate din coada de operatii */
{
 int i,j,t;
 for(i=0;i<maxQueue;i++){
   if(mesQueue[i].counter == counter)
     for(j=0;j<maxHosts;j++){
      for(t=0;t<maxAck;t++)
        if((ackList[t].sender == j) && (ackList[t].ts>mesQueue[i].ts )
                       && (ackList[t].counter == counter))
            break;
        if(t==maxAck)
            break;
     }
    if(j==maxHosts)
    return i;
 }
 return -1;
}

void RemoveQueue(int poz)
/* sterge elementul din pozitia poz din coada */
{
 int i;
 for(i = poz; i < maxQueue; i++)
   mesQueue[i] = mesQueue[i+1] ;
 maxQueue-- ;
}
int max(int a,int b)
{
 if(a > b)
  return a ;
 else
  return b ;
}

int CreateRemoteChan(void)
{
 int s, i;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     Panic(ERROR1);
 i = 1 ;
 if(setsockopt(s, SOL_SOCKET, SO_BROADCAST, (char *) &i,
sizeof(i)))
     Panic(ERROR7);
 bzero((char *)&sar, sizeof(sar)) ;
 sar.sin_family     = AF_INET ;
 sar.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sar.sin_port       = htons(BROADCAST_PORT) ;
 if(bind(s, (struct sockaddr *)&sar, sizeof(sar)) < 0)
     Panic(ERROR2);
 return s;
}

int CreateLocalChan(void)
{
 int s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     Panic(ERROR3);
 bzero((char *)&sal, sizeof(sal)) ;
 sal.sin_family     = AF_INET ;
 sal.sin_addr.s_addr = htonl(INADDR_ANY)  ;
 sal.sin_port       = htons(LOCAL_PORT) ;
 if(bind(s, (struct sockaddr *)&sal, sizeof(sal)) < 0)
     Panic(ERROR4);
 return s;
}

void CreateAnswerChan(void)
/* creaza si deschide answerChan*/
{
 unlink(FIFO_NAME);
 if( mknod(FIFO_NAME, S_IFIFO | PERMS , 0)<0)
     Panic(ERROR5);
 if((answerChan = open(FIFO_NAME, O_RDWR))<0)
     Panic(ERROR6);
}

void CloseAnswerChan(void)
 /* sterge pipe */
{
 unlink(FIFO_NAME);
}

void Debug(const char *c, ...)
{
  va_list args;
  va_start(args,c);
  vprintf(c, args);
  va_end(args);
}

void Panic( int err){
 printf("Exit:  %d error \n", err);
 exit(0);
}
/* ******************************************************************
       Functiile puse la dispozitia unui client
 care foloseste contori de eveniment distribuiti - varianta Unix
 -Suportul teoretic este prezentat in capitolul 4, la 4.3.4.2
 -Functiile care pot fi folosite sunt de un client :

 int  EventCountersAdvance(int contor)
 int EventCountersAwait(int contor, int val)
 int EventCountersRead(int contor)
 int EventCountersInitialize(void)
 void EventCountersEnd(void)
  - Comunicarea cu serverul  se face printr-un pipe FIFO
******************************************************************/
 #include "counteru.h"
 int  lChan;
 struct sockaddr_in sad;
 int answerChan;     /* handler  answerChan*/
 int GetServerAddress(void);
 int OpenAnswerChan(void);
 int  EventCountersAdvance(int counter)  /* Op. ADVANCED */
 {
  char info[LEN];
```


BUPT

```
int err;
 if (counter > MAX_CONTOR-1  ll counter < 0)
     return  -1;
sprintf(info,"%2d-%2d",ADVANCE,counter);
sendto(lChan,info,LEN,0,(struct sockaddr *)&sad,sizeof(sad));
 return 0;
}
int EventCountersAwait(int counter, int val) /*Op. AWAIT*/
{
 char info[LEN];
 if (counter > MAX_CONTOR-1  ll counter < 0)
     return  -1;
sprintf(info,"%2d-%2d-%2d",AWAIT,counter,val);
sendto(lChan,info,LEN,0,(struct sockaddr *)&sad, sizeof(sad));
 /* Asteptare   deblocare  */
read(answerChan,info,6);
return 0;
}
int EventCountersRead(int counter) /* OP. READ */
{
 char info[200];
 int val;
 if (counter > MAX_CONTOR-1  ll counter < 0)
       return  -1;
sprintf(info,"%2d-%2d",READ,counter) ;
sendto(lChan,info, LEN, 0,(struct sockaddr *) &sad,sizeof(sad)) ;
  read(answerChan,info,6);
 sscanf(info,"%d",&val) ;
return val;
}
int EventCountersInitialize(void)/*realizeaza conectarea la server*/
{
 int errCode;
 if ((errCode=OpenAnswerChan()) != 0 )
   return errCode;
 if ((errCode= GetServerAddress()) != 0 )
   return errCode;
 memset(&sad, 0, sizeof(sad)) ;
 sad.sin_family      = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(MyServerAddress) ;
 sad.sin_port        = htons(LOCAL_PORT) ;
 if((lChan= socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    return ERROR5;
 return 0;
}
void  EventCountersEnd(void){ /* deconectare de la server */
 close(lChan);
 unlink(FIFO_NAME);
}
int GetServerAddress(void) /*determina adr,serverului propriu*/
{
struct in_addr adr;
char HostName[MAX_HOST_NAME];
struct hostent *host;
 if (gethostname(HostName,MAX_HOST_NAME) != 0)
   return ERROR2;
 if((host = gethostbyname(HostName)) == NULL)
   return ERROR3;
 if(host != NULL)
   adr.s_addr = ((struct in_addr *)(host->h_addr_list[0]))->s_addr;
 if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
   return ERROR4;
 return 0;
}
 char * GetMyAddress(void){
  int errCode;
 if  ((errCode = GetServerAddress()) != 0)
    return NULL;
 else
   return MyServerAddress;
}
int OpenAnswerChan(void) /*  deschide answerChan */
{
 if((answerChan= open(FIFO_NAME, O_RDWR))<0)
    return ERROR1;
  return 0;
 }
```

B9. Implementarea secventiatorilor distribuiţi -varianta UNIX

```
/* ***********************************************************
         Server  pentru  secventiatori distribuiti
  -Serverul  este folosit in anexa B10  pentru  implementarea
excluderii reciproce cu ajutorul contorilor de evenimente distribuiti
si  a secventiatorilor distribuiti
- Suportul teoretic este prezentat in capitolul 4, la 4.3.4.3-
***********************************************************
*/#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdarg.h>
# include sequer.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define BROADCAST_PORT   6570
#define LOCAL_PORT   6571
#define FIFO_NAME  "answerChan"
#define PERMS 0666
#define MAX_MES_QUEUE  100 /* dim max a cozii de mesaje */
#define MAX_ACK_QLEN   1000 /* dim max a listei de confir.*/
#define ACKN      4
#define TICKET   5
#define LEN       100
#define TRUE      1
#define FALSE     0
#define DEBUG   TRUE


/* ************************************************************* **
 * Functii si variabile interne serverului                        *
************************************************************* ***/
typedef enum { TICK, ACK } kind ;
typedef struct  {/* struct. unui element din coada de op. neconf.*/
    int sender;          /* identificatorul emitatorului */
    int ts;              /* marca de timp */
}QUEUE;
typedef struct{   /* structura unui element din coada de Ack*/
    int sender;   /* identificatorul emitatorului */
    int ts;       /* marca de timp */
}ACKNOWLEDGE;int CreateRemoteChan(void);

int CreateLocalChan(void);
void CreateAnswerChan(void);
void CloseAnswerChan(void);
void InsertQueue(int sender, int ts);
void RemoveQueue(int poz);
void InsertAck(int sender, int ts);
int FindFullyAck(int sender);
void Debug(const char *c, ...);
int max(int a, int b);

int answerChan ;
QUEUE mesQueue[MAX_MES_QUEUE];
ACKNOWLEDGE ACKList[MAX_ACK_QLEN];
struct sockaddr_in  sar,sal;
int  sarl, sall;
int maxQueue, maxAck;
int maxHosts;
int myID;
int maxACK;
```


BUPT

```
int poz;
void main(argc,argv)
int argc;
char *argv[];
{
int rChan,lChan;
struct sockaddr_in sdr;
fd_set rfds;            /* read file descriptor set*/
fd_set afds;            /* active file descriptor set*/
char sBuf[LEN],rBuf[LEN];     /* buffere */
int localTS =0;         /* marca de timp locala */
int nfds, err;          /* variabile de lucru */
int nT=0;
int typeMes, ts,sender, recSender; /* componente in mesaje */
kind k;

/* *************************************************************** ***
faza de initializare: se prieia din linia de comanda identificatorul
host-ului local si numarul de procese.
Se creaza canalele.
Se pun in asteptare de mesaje canalele remoteChan, localChan
 *************************************************************** *** */
if(argc!=3) {
   Debug("Usage : [name] [identifier] [hosts] \n");
   exit(0);
}
CreateAnswerChan();
myID = atoi(argv[1]);
maxHosts = atoi(argv[2]);
rChan = CreateRemoteChan();
lChan = CreateLocalChan();
nfds = getdtablesize();
FD_ZERO(&afds);
FD_SET(rChan,&afds);
FD_SET(lChan,&afds);

/* *************************************************************** ***
faza de tratare a mesajelor primite de la clientul local
sau din retea, de la serverele de pe celelalte host-uri
 *************************************************************** *** */
while(TRUE){
 Debug("Waiting messages...\n");
 bcopy((char *)&afds, (char *)&rfds, sizeof(rfds));
 if(select(nfds,&rfds,(fd_set *)0,(fd_set *)0,(struct timeval *)0)<0)
    Debug("Select error \n");
 if(FD_ISSET(rChan,&rfds)){   /* mesaj din retea */
   sarl = sizeof(sar);
   err=recvfrom(rChan,rBuf,LEN,0,(struct sockaddr*)&sar,&sarl) ;
   if(err<0)
      Debug("Recvfrom - remote- error\n");
   scanf(rBuf,"%2d-%2d-%6d",&typeMes,&recSender,&ts);
   switch(typeMes){
   case 4: k=ACK;
             break;
   case 5: k=TICK;
            break;
  }
   localTS=max(localTS,ts+1);  /* actualizare marca de timp */
   localTS++;
  if(k==TICK){                 /* tratare TICK */
    InsertQueue(recSender,ts);
    memset(&sdr, 0, sizeof(sdr));
    sdr.sin_family     = AF_INET ;
    sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
    sdr.sin_port       = htons(BROADCAST_PORT) ;
    sprintf(sBuf,"%2d-%2d-%6d",ACKN,myID,localTS);
    Debug("Sending:%s\n",sBuf);  /* trimite confirmarea */
    sendto(rChan,sBuf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
    localTS++;
  }
  else
  if(k==ACK){                  /* tratare ACK */
    InsertAck(recSender,ts);
    poz = FindFullyAck(recSender);
    if(poz>=0){
      nT++;
       sender = mesQueue[poz].sender;
      RemoveQueue(poz);
      if(sender == myID) {
          sprintf(sBuf,"%d",nT);
           write(answerChan,sBuf,6) ;
      }
    }
  }
}
if(FD_ISSET(lChan,&rfds)){     /* mesaj de la clientul local */
  sall = sizeof(sal);
  err =recvfrom(lChan,rBuf,LEN,0,(struct sockaddr*)&sal, &sall);
  if(err<=0)
    Debug("Recvfrom - local -error\n");
  sscanf(rBuf,"%2d",&typeMes);
  switch(typeMes){
    case 5: k=TICK;
             break;
  }
  if(k==TICK){                  /* tratare TICK */
    memset(&sdr, 0, sizeof(sdr));
    sdr.sin_family     = AF_INET ;
    sdr.sin_addr.s_addr = inet_addr(BROADCAST_ADDR) ;
    sdr.sin_port       = htons(BROADCAST_PORT) ;
    sprintf(sBuf,"%2d-%2d-%6d",TICKET,myID,localTS);
    Debug("Sending:%s\n",sBuf);
    sendto(rChan,sBuf,LEN,0,(struct sockaddr*)&sdr,sizeof(sdr));
    localTS++;
  }
 }
}
}
void InsertQueue(int sender, int ts)
/* insereaza un element in coada astfel incat coada */
 /* sa ramana ordonata */
{
 int i,j;
 if(maxQueue == 0) {
    mesQueue[0].sender = sender ;
    mesQueue[0].ts = ts ;
     maxQueue = 1 ;
 }
 else {
    i = 0 ;
    while((i < maxQueue) && (ts >= mesQueue[i].ts))
             i++ ;
   for(j = maxQueue-1; j >= i; j--)
      mesQueue[j+1] = mesQueue[j];
   maxQueue++ ;
   mesQueue[i].sender = sender ;
  mesQueue[i].ts = ts ;
  }
}
void InsertAck(int sender, int ts)
{
  ACKList[maxACK].sender = sender ;
  ACKList[maxACK].ts = ts ;
  maxACK++;
 }
int FindFullyAck(int sender)
/* returneaza inregistrarile de tipul        */
/*  fully acknowledgement din coada     */
{
 int i,j,t;
 for(i=0;i<maxQueue;i++){
   for(j=0;j<maxHosts;j++){
     for(t=0;t<maxACK;t++)
      if((ACKList[t].sender==j)&&(ACKList[t].ts > mesQueue[i].ts ))
          break;
     if(t==maxACK)
          break;
    }
  if(j==maxHosts)
    return i;
 }
 return -1;
}
```


BUPT

```
void RemoveQueue(int poz)
/* sterge elementul din pozitia poz din coada */
{
 int i;
 for(i = poz; i < maxQueue; i++)
   mesQueue[i] = mesQueue[i+1] ;
 maxQueue-- ;
}

int max(int a, int b)
{
 if(a>=b)
    return a;
  else
    return b;
}

int CreateRemoteChan(void)
{
 int s, i;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR1);
 i = 1 ;
 if(setsockopt(s,SOL_SOCKET,SO_BROADCAST,(char *)&i,
                                        sizeof(i)))
    Panic(ERROR7);
 bzero((char *)&sar, sizeof(sar)) ;
 sar.sin_family      = AF_INET ;
 sar.sin_addr.s_addr = htonl(INADDR_ANY) ;
 sar.sin_port        = htons(BROADCAST_PORT) ;
 if(bind(s, (struct sockaddr *)&sar, sizeof(sar)) < 0)
    Panic(ERROR2);
 return s;
}

int CreateLocalChan(void)
{
 int s;
 if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    Panic(ERROR3);
 bzero((char *)&sal, sizeof(sal)) ;
 sal.sin_family      = AF_INET ;
 sal.sin_addr.s_addr = htonl(INADDR_ANY) ;
 sal.sin_port        = htons(LOCAL_PORT) ;
 if(bind(s, (struct sockaddr *)&sal, sizeof(sal)) < 0)
    Panic(ERROR4);
 return s;
}

void CreateAnswerChan(void)
/* creaza si deschide answerChan*/
{
 unlink(FIFO_NAME);
 if( mknod(FIFO_NAME, S_IFIFO | PERMS , 0)<0)
    Panic(ERROR5);
 if((answerChan = open(FIFO_NAME, O_RDWR))<0)
    Panic(ERROR6);
}

void CloseAnswerChan(void)
/* sterge pipe */
{
 unlink(FIFO_NAME);
}

void Debug(const char *c, ...)
{
 va_list args;
 va_start(args,c);
 vprintf(c, args);
 va_end(args);
}
void Panic( int err){
 printf("Exit:  %d error \n", err);
 exit(0);
}
```

```
/* ***************************************************************
      Functiile puse la dispozitia unui client
 care foloseste  secventiatori distribuiti - varianta  Unix
 -Suportul teoretic este prezentat in capitolul 4, la 4.3.4.3
 -Functiile care pot fi folosite sunt de un client :
int SequencerInitialize(void)
int  SequencerTick(ivoid)

 - Comunicarea cu serverul  se face printr-un pipe FIFO
****************************************************************/
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "tickeru.h"

#define BROADCAST_ADDR  "193.226.37.255"
#define LOCAL_PORT  6571
#define FIFO_NAME  "answerChan"
#define PERMS 0666
#define LEN     100
#define MAX_HOST_NAME   256
#define TICKET 5
#define MAX_FIFO_LEN 6

int GetServerAddress(void);
int  SequencersInitialize(void);
int  SequencerTicket(void);
int  SequencerEnd(void);
int OpenAnswerChan(void) ;

int    lChan ;
struct sockaddr_in sad;
int answerChan ;

int  SequencerTicket(void) /*determinare valoare secventiator */
{
char info[LEN];
int val;
sprintf(info,"%2d",TICKET) ;
sendto(lChan,info,LEN, 0,(struct sockaddr *) &sad,sizeof(sad)) ;
read(answerChan,info,MAX_FIFO_LEN);
sscanf(info,"%d",&val);
return val;
}

int  SequencersInitialize(void)/*realizeaza conectarea la server*/
{
 int errCode;
 if ((errCode=OpenAnswerChan()) != 0 )
   return errCode;
 if ((errCode= GetServerAddress()) != 0 )
   return errCode;
 memset(&sad, 0, sizeof(sad)) ;
 sad.sin_family      = AF_INET ;
 sad.sin_addr.s_addr = inet_addr(MyServerAddress) ;
 sad.sin_port        = htons(LOCAL_PORT) ;
 if((lChan= socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    return ERROR5;
 return 0;
}
int  SequencerEnd(void){ /* deconectare de la server */
 close(lChan);
 unlink(FIFO_NAME);
}

 char * GetMyAddress(void){
 int errCode;
```


BUPT

```
 if  ((errCode = GetServerAddress()) != 0)
    return NULL;
 else
   return MyServerAddress;
}

int GetServerAddress(void) /*determina adr,serverului propriu*/
{
struct in_addr adr;
char HostName[MAX_HOST_NAME];
struct hostent *host;
 if (gethostname(HostName,MAX_HOST_NAME) != 0)
    return ERROR2;
 if((host = gethostbyname(HostName)) == NULL)
    return ERROR3;
 if(host != NULL)
    adr.s_addr = ((struct in_addr *)(host->h_addr_list[0]))->s_addr;
 if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
    return ERROR4;
 return 0;
}

int OpenAnswerChan(void) /*  deschide answerChan */
{
 if((answerChan= open(FIFO_NAME, O_RDWR))<0)
    return ERROR1;
  return 0;
 }
```

B10. Implementarea excluderii reciproce folosind secvenţiatori şi contori de evenimente distribuiţi varianta Unix

```
/* ******************************************************************
 Se folosesc cele doua servere prezentate anterior:
cel pentru secventiatori distribuiti si cel pentru contori de
evenimente distribuiti
Functiile puse la dispozitia programelor client sunt::

int  EventCounterAdvanced(int contor)
int EventCounterAwait(int contor, int val)
int EventCounterRead(int contor)
int  SequencerTicket(void)
char * GetMyAddress(void)
int ExclAccessInitialize(void);
void ExclAccessEnd(void);
**** *************************************************************/
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "evsequer.h"

 #define LOCAL_PORT_EVENT   6561
 #define LOCAL_PORT_SEQUENCER 6571
 #define FIFO_NAMEE  "eventChan"
 #define FIFO_NAMET  "sequencerChan"
 # define MAX_CONTOR 10
 #define MAX_SERVER_ADDR 20
 #define LEN 400
 #define MAX_FIFO_LEN    6
 #define MAX_HOST_NAME 256
 #define ADVANCE 1
 #define AWAIT   2
 #define READ    3
 #define TICKET  5

char MyServerAddress[MAX_SERVER_ADDR];
int  SequencerTicket(void);
int EventCounterAdvance(int);
int EvemtCounterAwait(int, int);
int  EventCounterRead(int);
int GetServerAddress(void);
int ExclAccessInitialize(void);
void ExclAccessEnd(void);
int OpenAnswerChannels(void);

int  IChanE,IChanT ;
int  answerChanE,answerChanT ;
struct sockaddr_in saa,sat;

int  SequencerTicket(void)
/*Operatie de TICKET*/
{
 char info[LEN];
 int val;
 sprintf(info,"%2d",TICKET) ;
 sendto(IChanT,info, LEN, 0,(struct sockaddr *) &sat,sizeof(sat)) ;
 read(answerChanT,info,MAX_FIFO_LEN);
 sscanf(info,"%d",&val); .
 return val;
}

int  EventCounterAdvanced(int contor)
/* Operatie de ADVANCED -  */
{
char bf[LEN];
if (contor > MAX_CONTOR-1  II contor < 0)
     return  -1;
sprintf(bf,"%2d-%2d",ADVANCE,contor);
sendto(IChanE,bf,LEN,0,(struct sockaddr*)&saa,sizeof(saa));
return 0;
}

int  EventCounterAwait(int contor, int val)
/*Operatie de AWAIT*/
{
 char bf[LEN];
if (contor > MAX_CONTOR-1  II contor < 0)
     return  -1;
sprintf(bf,"%2d-%2d-%2d",AWAIT,contor,val);
sendto(IChanE,bf, LEN, 0,(struct sockaddr *)&saa, sizeof(saa));
read(answerChanE,bf,MAX_FIFO_LEN);
return 0;
}

int EventCounterRead(int contor)
/*Operatie de READ*/
{
 char bf[ LEN ];
 int val;
 if (contor > MAX_CONTOR-1  II contor < 0)
       return  -1;
 sprintf(bf,"%2d-%2d",READ,contor) ;
 sendto(IChanE,bf, LEN, 0,(struct sockaddr *) &saa,sizeof(saa)) ;
  read(answerChanE,bf,MAX_FIFO_LEN);
 sscanf(bf,"%d",&val) ;
 return val;
}

 char * GetMyAddress(void){
  int errCode;
  if  ((errCode = GetServerAddress()) != 0)
     return NULL;
  else
    return MyServerAddress;
 }

 int GetServerAddress(void) /*determina adr,serverului propriu*/
 {
 struct in_addr adr;
 char HostName[MAX_HOST_NAME];
 struct hostent *host;
  if (gethostname(HostName,MAX_HOST_NAME) != 0)
```


BUPT

```
  return ERROR3;
if((host = gethostbyname(HostName)) == NULL)
  return ERROR4;
if(host != NULL)
  adr.s_addr = ((struct in_addr *)(host->h_addr_list[0]))->s_addr;
if(strcpy(MyServerAddress,inet_ntoa(adr)) == NULL)
  return ERROR5;
return 0;
}

int ExclAccessInitialize(void)
{
 int errCode;
 if ((errCode =OpenAnswerChannels()) != 0)
   return errCode;
 if ((errCode=GetServerAddress()) !=0 )
  return errCode;
 memset(&saa, 0, sizeof(saa)) ;
 saa.sin_family      = AF_INET ;
 saa.sin_addr.s_addr = inet_addr(MyServerAddress) ;
 saa.sin_port        = htons(LOCAL_PORT_EVENT) ;
 if((lChanE = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   return  ERROR6;
 memset(&sat, 0, sizeof(sat)) ;
 sat.sin_family      = AF_INET ;
 sat.sin_addr.s_addr = inet_addr(MyServerAddress) ;
 sat.sin_port        = htons(LOCAL_PORT_SEQUENCER) ;
 if((lChanT = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   return ERROR7;
 return 0;

}

int  OpenAnswerChannels()
/* creaza si deschide  canalele answer*/
{
 if((answerChanE = open(FIFO_NAMEE, O_RDWR))<0)
   return ERROR1;
  if((answerChanT = open(FIFO_NAMET, O_RDWR))<0)
   return ERROR2;
 return 0;
 }

void ExclAccessEnd(void)
 /* sterge canalele answer */
{
 unlink(FIFO_NAMEE);
 unlink(FIFO_NAMET);
 close(lChanE);
 close(lChanT);
}
/* ******************************************************************** ***

                    Programul client

/ ******************************************************************** *** */
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <X11/Xaw/List.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Dialog.h>
#include <X11/Xaw/AsciiText.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <signal.h>
#include "adtic.h"
#define LEN 100

Widget
  widQuit,widInfo,widView, widPaned,widBoxMenu,
  widBoxView, widP, widV, widTopLevel,
  widRead,widRequest,widRelease,widInit,label;

int eventNo=1,eventVal;

void CallbackInit
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackInfo
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackRead
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackRequest
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackRelease
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackQuit
        (Widget w,XtPointer client_data,XtPointer call_data);

main(argc, argv)
int  argc;
char **argv;
{
 XtAppContext app_context;
 if (ExclAccessInitialize())
        exit(1);
 widTopLevel = XtVaAppInitialize(
         &app_context,
        "Main",
        NULL,0,
        &argc, argv,
        NULL,
        NULL);
 widPaned = XtVaCreateManagedWidget(
         "mainPaned",
        panedWidgetClass,
        widTopLevel,
        NULL) ;
 widBoxMenu = XtVaCreateManagedWidget(
        "boxMenu",
         boxWidgetClass,
         widPaned,
         XtNorientation, XtorientHorizontal,
         NULL);
  widInfo = XtVaCreateManagedWidget(
        "Info",
        commandWidgetClass,
        widBoxMenu,
        NULL);
widInit = XtVaCreateManagedWidget(
        "Init",
        commandWidgetClass,
        widBoxMenu,
        NULL);
widRead = XtVaCreateManagedWidget(
        "Read",
        commandWidgetClass,
        widBoxMenu,
        NULL);
widRequest = XtVaCreateManagedWidget(
        "Request",
        commandWidgetClass,
        widBoxMenu,
        NULL);
widRelease = XtVaCreateManagedWidget(
        "Release",
        commandWidgetClass,
        widBoxMenu,
        NULL);
 widQuit = XtVaCreateManagedWidget(
        "Quit",
        commandWidgetClass,
        widBoxMenu,
        NULL);
 label = XtVaCreateManagedWidget(
        "label",
```


BUPT

```
                labelWidgetClass,
                widPaned,
               XtNlabel,"\n                        \n",
               XtNwidth,400,
                NULL);
 XtAddCallback(widInfo, XtNcallback, CallbackInfo, 0);
 XtAddCallback(widQuit, XtNcallback, CallbackQuit, 0);
 XtAddCallback(widInit, XtNcallback, CallbackInit, 0);
 XtAddCallback(widRead, XtNcallback, CallbackRead, 0);
 XtAddCallback(widRequest, XtNcallback, CallbackRequest, 0);
 XtAddCallback(widRelease, XtNcallback, CallbackRelease, 0);
 XtRealizeWidget(widTopLevel);
 XtAppMainLoop(app_context);
}

void CallbackQuit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
  ExclAccessEnd();
  exit(0);
}

void CallbackInfo( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[LEN],*adr;
 adr=GetMyAddress();
 sprintf(msg,"Server address  %s",adr);
 XtVaSetValues(label,XtNlabel,msg,NULL);
}

void CallbackInit( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[LEN];
 int i;
 EventCounterAdvanced(eventNo);
 sprintf(msg," Counter = %d",eventNo);
 XtVaSetValues(label,XtNlabel,msg,NULL);
}

void CallbackRead( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[LEN];
 int i;
 i =  EventCounterRead(eventNo);
 sprintf(msg,"%d",i);
XtVaSetValues(label,XtNlabel,msg,NULL);
}

void CallbackRequest( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[LEN];
 eventVal = Ticket();
 EventCounterAwait(eventNo, eventVal);
  XtVaSetValues(label,XtNlabel,"I am in critical section...",NULL);
}

void CallbackRelease( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[LEN];
  EventCounterAdvanced(eventNo);
 XtVaSetValues(label,XtNlabel,"Done release...",NULL);
 }
```


BUPT

ANEXA C
Implementarea unui toolkit pentru controlul accesului concurent la o bază de date în rețele Novell, Unix și Windows

C1. Implementarea primei soluții bazate pe semafoare în rețele Novell

```
/*********************************************************************************************************
 Suportul teoretic este prezentat in capitolul 5, la 5.1.1.
 In prima solutie se folosesc liste de identificatori ai proceselor clienti care sunt gestionate FIFO
 Procesele care cer acces exclusiv sunt mai prioritare
 Functiilor  care pot fi apelate de programele utilizator sunt:

int ExclOrSharedAccInitialize(void);     /*  realizeaza  conectarea la sistem */
void ExclOrSharedAccessEnd(void);        /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );             /* realizeaza obtinerea accesului pentru operatii partajabile asupra  a bazei  da date */
int CloseSharedAccess(void );            /* realizeaza terminarea accesului i partajabil asupra  a bazei  da date */
int OpenExclAccess(void );               /* realizeaza obtinerea accesului pentru operatii exclusive asupra  a bazei  da date */
int CloseExclAccess(void );              /* realizeaza terminarea accesului exclusiv asupra  a bazei  da date */
*********************************************************************************************************/
#include <nit.h>
#include <stdlib.h>
#include <string.h>

#define SHARED_ACCESS 1                  /* codficarea celor doua moduri de lucru: shereable, exclusive */
#define EXCL_ACCESS 2
#define TIMEOUT 360                      /* timpul de asteptare maxim la un semafor */
#define MAX_CON 50                       /* numarul maxim de noduri care pot participa la algoritm */
#define MAX_LENGTH 30                    /* lungimea unui mesaj */
#define TRUE 1
#define FALSE 0
int ExclOrSharedAccessInitialize(void);  /*  realizeaza  conectarea la sistem */
void ExclOrSharedAccessEnd(void);        /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );             /* realizeaza obtinerea accesului pentru operatii partajabile asupra  a bazei  da date */
int CloseSharedAccess(void );            /* realizeaza terminarea accesului i partajabil asupra  a bazei  da date */
int OpenExclAccess(void );               /* realizeaza obtinerea accesului pentru operatii exclusive asupra  a bazei  da date */
int CloseExclAccess(void );              /* realizeaza terminarea accesului exclusiv asupra  a bazei  da date */
int GetPriorityListProcess(int );        /* functie apelabila local; nu este pusa la dispozitia clientilor */

long hNoSharedAccess, hNoExclAccess, hInExclAccess;      /* handlere pentru semafoare */
long hSharedDataProtect, hWaitingSharedAccess[MAX_CON+1],hWaitingExclAccess[MAX_CON+1], hFlag[MAX_CON+1];
WORD pNoSharedAccess, pNoExclAccess, pInExclAccess;
WORD pSharedDataProtect, pWaitingSharedAccess[MAX_CON+1], pWaitingExclAccess[MAX_CON+1], pFlag[MAX_CON+1];
int vNoSharedAccess, vNoExclAccess, vInExclAccess, vFlag[MAX_CON+1];
int ConnectionNumber;
int code;
char name1[MAX_LENGTH] = "WaitingSharedAccess";        /* nume pentru semafoarele utilizate */
char name2[MAX_LENGTH] = "WaitingExclAccess";
char name3[MAX_LENGTH] = "Flag";
char temp[2] = "A";

int ExclOrSharedAccessInitialize(void)                   /*  realizeaza  conectarea la sistem */
{
 int i;
 if ((code = OpenSemaphore("NoSharedAccess",0, &hNoSharedAccess, &pNoSharedAccess)) != 0 ) return 1;
 if ((code = OpenSemaphore("NoExclAccess", 0, &hNoExclAccess, &pNoExclAccess)) != 0) return 2;
 if ((code = OpenSemaphore("InExclAccess",1, &hInExclAccess, &pInExclAccess)) != 0) return 3;
 if ((code = OpenSemaphore("SharedDataProtect",1, &hSharedDataProtect, &pSharedDataProtect)) != 0) return 4;
 for (i=1;i<=MAX_CON;i++){
            temp[0] = 'A'+i;
  if ((code = OpenSemaphore((char *)strcat(name1,temp),0,&hWaitingSharedAccess[i], &pWaitingSharedAccess[i])) != 0) return 5;
   strcpy(name1,"WaitingSharedAccess");
   if ((code = OpenSemaphore((char *)strcat(name2,temp),0,&hWaitingExclAccess[i], &pWaitingExclAccess[i]))  != 0) return 6;
   strcpy(name2,"WaitingExclAccess");
   if ((code = OpenSemaphore((char *)strcat(name3,temp),0,&hFlag[i], &pFlag[i])) != 0) return 7;
   strcpy(name3,"Flag");
 }
 ConnectionNumber = (int)GetConnectionNumber();
 return 0;
}
```


BUPT

```
void ExclOrSharedAccessEnd(void)        /* realizeaza deconectarea de la sistem */
{
int i;
 CloseSemaphore(hNoSharedAccess);    /* inchide semafoarele utilizate */
 CloseSemaphore(hNoExclAccess);
 CloseSemaphore(hSharedDataProtect);
 CloseSemaphore(hInExclAccess);
 for (i=1;i<=MAX_CON;i++){
   CloseSemaphore(hWaitingSharedAccess[i]);
   CloseSemaphore(hWaitingExclAccess[i]);
   CloseSemaphore(hFlag[i]);
 }
}

int OpenSharedAccess(void)              /* realizeaza obtinerea accesului pentru operatii partajabile asupra  a bazei  da date */
{
  if ((code = WaitOnSemaphore(hSharedDataProtect,TIMEOUT)) != 0) return 8;
  if ((code = ExamineSemaphore(hNoExclAccess,&vNoExclAccess,&pNoExclAccess)) != 0) return 9;
  if (vNoExclAccess == 0) {
    if ((code = SignalSemaphore(hNoSharedAccess)) != 0) return 10;
    if ((code = SignalSemaphore(hWaitingSharedAccess[ConnectionNumber])) != 0) return 11;
  }
  else if ((code = SignalSemaphore(hFlag[ConnectionNumber])) != 0 ) return 12;
  if ((code = SignalSemaphore(hSharedDataProtect)) != 0)return 13;
  if ((code = WaitOnSemaphore(hWaitingSharedAccess[ConnectionNumber],TIMEOUT)) != 0) return 14;
  return 0;
}

int CloseSharedAccess(void)             /* realizeaza terminarea accesului i partajabil asupra  a bazei  da date */
{
  int k;
  if ((code = WaitOnSemaphore(hSharedDataProtect,TIMEOUT)) != 0) return 15;
  if ((code = WaitOnSemaphore(hNoSharedAccess,TIMEOUT)) != 0) return 16;
  if ((code = ExamineSemaphore(hNoSharedAccess,&vNoSharedAccess,&pNoSharedAccess)) != 0) return 17;
  if ((code = ExamineSemaphore(hNoExclAccess,&vNoExclAccess,&pNoExclAccess)) != 0) return 18;
  if ((vNoSharedAccess == 0) && (vNoExclAccess > 0)) {
    k = GetPriorityListProcess(EXCL_ACCESS);
    if (k < 0) return abs(k);
    if ((code = WaitOnSemaphore(hInExclAccess, TIMEOUT)) != 0) return 19;
    if ((code = SignalSemaphore(hWaitingExclAccess[k])) != 0) return 20;
  }
  if ((code = SignalSemaphore(hSharedDataProtect)) != 0) return 21;
  return 0;
}

int OpenExclAccess(void)      /* realizeaza obtinerea accesului pentru operatii exclusive asupra  a bazei  da date */
{
  if ((code = WaitOnSemaphore(hSharedDataProtect,TIMEOUT)) != 0) return 22;
  if ((code = SignalSemaphore(hNoExclAccess)) != 0) return 23;
  if ((code = ExamineSemaphore(hNoSharedAccess,&vNoSharedAccess,&pNoSharedAccess)) !=0) return 24;
  if ((code = ExamineSemaphore(hInExclAccess,&vInExclAccess,&pInExclAccess)) != 0) return 25;
  if ((vNoSharedAccess==0) && (vInExclAccess==1)) {
     if ((code = WaitOnSemaphore(hInExclAccess, TIMEOUT)) != 0) return 26;
     if ((code = SignalSemaphore(hWaitingExclAccess[ConnectionNumber])) != 0) return 27;
  }
  else {
     if ((code = SignalSemaphore(hFlag[ConnectionNumber])) != 0) return 28;
     if ((code = SignalSemaphore(hFlag[ConnectionNumber])) != 0) return 29;
  }
  if ((code = SignalSemaphore(hSharedDataProtect)) != 0) return 30;
  if ((code = WaitOnSemaphore(hWaitingExclAccess[ConnectionNumber],TIMEOUT)) != 0) return 31;
  return 0;
}

int CloseExclAccess(void)      /* realizeaza terminarea accesului exclusiv asupra  a bazei  da date */
{
  int k;
  if ((code = WaitOnSemaphore(hSharedDataProtect,TIMEOUT)) != 0) return 32;
  if ((code = WaitOnSemaphore(hNoExclAccess,TIMEOUT)) != 0) return 33;
  if ((SignalSemaphore(hInExclAccess)) != 0) return 34;
  if ((ExamineSemaphore(hNoExclAccess,&vNoExclAccess,&pNoExclAccess)) != 0) return 35;
  if (vNoExclAccess > 0){
    k = GetPriorityListProcess(EXCL_ACCESS);
    if (k < 0) return abs(k);
    if ((code = WaitOnSemaphore(hInExclAccess,TIMEOUT)) != 0) return 36;
    if ((code = SignalSemaphore(hWaitingExclAccess[k])) != 0) return 37;
  }
```


BUPT

```
  else
    while((k = GetPriorityListProcess(SHARED_ACCESS)) > 0){
     if (k < 0) return abs(k);
     if ((code = SignalSemaphore(hNoSharedAccess)) != 0) return 38;
     if ((SignalSemaphore(hWaitingSharedAccess[k])) != 0)  return 39;
    }
  if ((code = SignalSemaphore(hSharedDataProtect)) != 0) return 40;
  return 0;
}

int GetPriorityListProcess(int type)          /* extrage din lista de  identificatori ai clientilor urmatorul client pentru a fi deblocat */
{
 int i;
 if(type == SHARED_ACCESS)            /* extrage un client pentru acces partajabil */
   for(i=1;i<=MAX_CON; i++){
          if ((code = ExamineSemaphore(hFlag[i],&vFlag[i],&pFlag[i])) != 0) return -41;
          if(vFlag[i]==1){
                    if ((code = WaitOnSemaphore(hFlag[i],TIMEOUT)) != 0) return -42;
                    return i;
          }
   }
 else                                /*extrage un client pentru acces exclusiv */
   for(i=1;i<=MAX_CON; i++){
    if ((code = ExamineSemaphore(hFlag[i],&vFlag[i],&pFlag[i])) != 0) return -43;
    if(vFlag[i]==2){
                    if (( code = WaitOnSemaphore(hFlag[i], TIMEOUT)) != 0) return -44;
                    if (( code = WaitOnSemaphore(hFlag[i], TIMEOUT)) != 0) return -44;
                    return i;
    }
   }
 return 0;
}
```

C2. Implementarea celei de-a doua soluţii bazate pe semafoare în reţele Novell

```
/************************************************************************************************************************
 Suportul teoretic este prezentat in capitolul 5, la 5.1.1.
 Procesele care cer acces exclusiv sunt mai prioritare.
 Functiilor care pot fi apelate de programele utilizator sunt:

int ExclOrSharedAccessInitialize(void);    /* realizeaza conectarea la sistem */
void ExclOrSharedAccessEnd(void);          /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );               /* realizeaza obtinerea accesului pentru operatii partajabile asupra a bazei da date */
int CloseSharedAccess(void );              /* realizeaza terminarea accesului i partajabil asupra a bazei da date */
int OpenExclAccess(void );                 /* realizeaza obtinerea accesului pentru operatii exclusive asupra a bazei da date */
int CloseExclAccess(void );                /* realizeaza terminarea accesului exclusiv asupra a bazei da date */

************************************************************************************************************************/
#include <nit.h>
#include <stdlib.h>

#define TIMEOUT 360                        /* timpul de asteptare maxim la un semafor */
#define FALSE 0
#define TRUE 1

int ExclOrSharedAccessInitialize(void);    /* realizeaza conectarea la sistem */
void ExclOrSharedAccessEnd(void);          /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );               /* realizeaza obtinerea accesului pentru operatii partajabile asupra a bazei da date */
int CloseSharedAccess(void );              /* realizeaza terminarea accesului i partajabil asupra a bazei da date */
int OpenExclAccess(void );                 /* realizeaza obtinerea accesului pentru operatii exclusive asupra a bazei da date */
int CloseExclAccess(void );                /* realizeaza terminarea accesului exclusiv asupra a bazei da date */

int WaitWithCondition(long hExclusive, long hWaiting, long hCounter); /* functii apelabile local; nu sunt pusa la dispozitia clientilor */
int SignalWithCondition(long hExclusive, long hWaiting, long hCounter);

long hNoSharedAccess, hNoExclAccess, hStopExclAccess, hStopSharedAccess;  /* handlere pentru semafoare */
long hExclusive, hExclDataProtect, hSharedDataProtect;                     /* handlere pentru semafoare */
WORD pNoSharedAccess, pNoExclAccess, pStopExclAccess, pStopSharedAccess;
WORD pExclusive, pExclDataProtect, pSharedDataProtect;
int  vNoSharedAccess, vNoExclAccess;
int code;
```


BUPT

```
int ExclOrSharedAccessInitialize(void)      /* realizeaza  conectarea la sistem */
{
 if ((code = OpenSemaphore("Exclusive", 1, &hExclusive, &pExclusive)) != 0) return 1;
 if ((code = OpenSemaphore("NoSharedAccess", 0, &hNoSharedAccess, &pNoSharedAccess)) != 0) return 2;
 if ((code = OpenSemaphore("NoExclAccess", 0, &hNoExclAccess, &pNoExclAccess)) != 0) return 3;
 if ((code = OpenSemaphore("SharedDataProtect", 1, &hSharedDataProtect, &pSharedDataProtect)) != 0) return 4;
 if ((code = OpenSemaphore("ExclDataProtect", 1, &hExclDataProtect, &pExclDataProtect)) != 0) return 5;
 if ((code = OpenSemaphore("StopExclAccess",1, &hStopExclAccess, &pStopExclAccess)) != 0) return 6;
 if ((code = OpenSemaphore("StopSharedAccess", 1,&hStopSharedAccess, &pStopSharedAccess)) != 0) return 7;
 return 0;
}
void ExclOrSharedAccessEnd(void)        /* realizeaza deconectarea de la sistem */
{
 CloseSemaphore(hExclusive);
 CloseSemaphore(hNoSharedAccess);
 CloseSemaphore(hNoExclAccess);
 CloseSemaphore(hSharedDataProtect);
 CloseSemaphore(hExclDataProtect);
 CloseSemaphore(hStopSharedAccess);
 CloseSemaphore(hStopExclAccess);
}
int OpenSharedAccess(void)              /* realizeaza obtinerea accesului pentru operatii partajabile asupra  a bazei  da date */
{
 int val;
 if ((code = WaitOnSemaphore(hExclusive, TIMEOUT)) != 0) return 8;
 if ((code = WaitOnSemaphore(hStopSharedAccess, TIMEOUT)) != 0) return 9;
 if ((val = WaitWithCondition(hSharedDataProtect,hStopExclAccess,hNoSharedAccess)) != 0)
     return val;
 if ((code = SignalSemaphore(hStopSharedAccess)) != 0) return 10;
 if ((code = SignalSemaphore(hExclusive)) != 0) return 11;
 return 0;
}

int CloseSharedAccess(void)             /* realizeaza terminarea accesului i partajabil asupra  a bazei  da date */
{
  return SignalWithCondition(hSharedDataProtect,hStopExclAccess,hNoSharedAccess);
}

int OpenExclAccess(void)                /* realizeaza obtinerea accesului pentru operatii exclusive asupra  a bazei  da date */
{
 int val;
 if ((val = WaitWithCondition(hExclDataProtect,hStopSharedAccess,hNoExclAccess)) != 0)
              return val;
 if ((code = WaitOnSemaphore(hStopExclAccess, TIMEOUT)) != 0) return 12;
 return 0;
}

int CloseExclAccess(void)               /* realizeaza terminarea accesului exclusiv asupra  a bazei  da date */
{
 if ((code = SignalSemaphore(hStopExclAccess)) != 0) return 13;
 return SignalWithCondition(hExclDataProtect,hStopSharedAccess,hNoExclAccess);
}

int WaitWithCondition(long hExclusive, long hWaiting, long hCounter)
{
 WORD pCounter;
 int vCounter;

 if ((code = WaitOnSemaphore(hExclusive, TIMEOUT)) != 0) return 14;
 if ((code = ExamineSemaphore(hCounter,&vCounter, &pCounter)) != 0) return 15;
 if (vCounter == 0)   if ((code = WaitOnSemaphore(hWaiting, TIMEOUT)) != 0) return 16;
 if ((code = SignalSemaphore(hCounter)) != 0) return 17;
 if ((code = SignalSemaphore(hExclusive)) != 0) return 18;
 return 0;
}
int SignalWithCondition(long hExclusive,long hWaiting, long hCounter)
{
 WORD pCounter;
 int vCounter;

 if ((code = WaitOnSemaphore(hExclusive, TIMEOUT)) != 0) return 19;
 if ((code = WaitOnSemaphore(hCounter, TIMEOUT)) != 0) return 20;
 if ((code = ExamineSemaphore(hCounter,&vCounter, &pCounter)) != 0) return 21;
 if (vCounter == 0)   if ((code = SignalSemaphore(hWaiting)) != 0) return 22;
 if ((code = SignalSemaphore(hExclusive)) != 0) return 23;
 return 0;
}
```


BUPT

C3. Implementarea celei de-a treia soluţii bazate pe semafoare în reţele Novell

```
/*****************************************************************************************************************
 Suportul teoretic este prezentat in capitolul 5, la 5.1.1.
 Procesele care cer acces partajat sunt mai prioritare.
 Functiilor care pot fi apelate de programele utilizator sunt:
int ExclOrSharedAccessInitialize(void);   /* realizeaza conectarea la sistem */
void ExclOrSharedAccessEnd(void);         /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );              /* realizeaza obtinerea accesului pentru operatii partajabile asupra a bazei da date */
int CloseSharedAccess(void );             /* realizeaza terminarea accesului i partajabil asupra a bazei da date */
int OpenExclAccess(void );                /* realizeaza obtinerea accesului pentru operatii exclusive asupra a bazei da date */
int CloseExclAccess(void );               /* realizeaza terminarea accesului exclusiv asupra a bazei da date */
*****************************************************************************************************************/
#include <nit.h>
#include <stdlib.h>
#define TIMEOUT 360                       /* timpul de asteptare maxim la un semafor */
#define FALSE 0
#define TRUE 1

int ExclOrSharedAccessInitialize(void);   /* realizeaza conectarea la sistem */
void ExclOrSharedAccessEnd(void);         /* realizeaza deconectarea de la sistem */
int OpenSharedAccess(void );              /* realizeaza obtinerea accesului pentru operatii partajabile asupra a bazei da date */
int CloseSharedAccess(void );             /* realizeaza terminarea accesului i partajabil asupra a bazei da date */
int OpenExclAccess(void );                /* realizeaza obtinerea accesului pentru operatii exclusive asupra a bazei da date */
int CloseExclAccess(void );               /* realizeaza terminarea accesului exclusiv asupra a bazei da date */

long hNoSharedAccess,hPrioritySharedAccess,hWaitingExclAccess,hExclusive;
WORD pNoSharedAccess,pExclusive,pWaitingExclAccess,pPrioritySharedAccess;
int vNoSharedAccess;
int code;

int ExclOrSharedAccessInitialize(void)    /* realizeaza conectarea la sistem */
{
 if ((code = OpenSemaphore("NoSharedAccess",0,&hNoSharedAccess,&pNoSharedAccess)) != 0) return 1;
 if ((code = OpenSemaphore("Exclusive",1,&hExclusive,&pExclusive)) != 0) return 2;
 if ((code = OpenSemaphore("WaitingExclAccess",1,&hWaitingExclAccess,&pWaitingExclAccess)) != 0) return 3;
 if ((code = OpenSemaphore("PrioritySharedAccess",1,&hPrioritySharedAccess,&pPrioritySharedAccess)) != 0) return 4;
 return 0;
}
void ExclOrSharedAccessEnd(void)          /* realizeaza deconectarea de la sistem */
{
 CloseSemaphore(hNoSharedAccess);
 CloseSemaphore(hExclusive);
 CloseSemaphore(hWaitingExclAccess);
 CloseSemaphore(hPrioritySharedAccess);
}
int OpenSharedAccess(void)                /* realizeaza obtinerea accesului pentru operatii partajabile asupra a bazei da date */
{
 if ((code = WaitOnSemaphore(hExclusive,TIMEOUT)) != 0) return 5;
 if ((code = SignalSemaphore(hNoSharedAccess)) != 0) return 6;
 if ((code = ExamineSemaphore(hNoSharedAccess, &vNoSharedAccess, &pNoSharedAccess)) != 0) return 7;
 if (vNoSharedAccess == 1)   if ((code = WaitOnSemaphore(hPrioritySharedAccess,TIMEOUT)) != 0) return 8;
 if ((SignalSemaphore(hExclusive)) != 0) return 9;
 return 0;
}
int CloseSharedAccess(void)               /* realizeaza terminarea accesului i partajabil asupra a bazei da date */
{
 if ((code = WaitOnSemaphore(hExclusive,TIMEOUT)) != 0) return 10;
 if ((code = WaitOnSemaphore(hNoSharedAccess,TIMEOUT)) != 0) return 11;
 if ((code = ExamineSemaphore(hNoSharedAccess, &vNoSharedAccess, &pNoSharedAccess)) != 0) return 12;
 if (vNoSharedAccess == 0) if ((code = SignalSemaphore(hPrioritySharedAccess)) != 0) return 13;
 if ((code = SignalSemaphore(hExclusive)) != 0) return 14;
 return 0;
}
int OpenExclAccess(void)                  /* realizeaza obtinerea accesului pentru operatii exclusive asupra a bazei da date */
{
 if ((code = WaitOnSemaphore(hWaitingExclAccess,TIMEOUT)) != 0) return 15;
 if ((code = WaitOnSemaphore(hPrioritySharedAccess,TIMEOUT)) != 0) return 16;
 return 0;
}
int CloseExclAccess(void)                 /* realizeaza terminarea accesului exclusiv asupra a bazei da date */
{
 if ((code = SignalSemaphore(hPrioritySharedAccess)) != 0) return 17;
 if ((code = SignalSemaphore(hWaitingExclAccess)) != 0) return 18;
 return 0;
}
```


BUPT

C4. Implementarea soluţiei bazate pe un proces coordonator în reţele Novell

```
/*****************************************************************/
 Suportul teoretic este prezentat in capitolul 5, la 5.1.2.
  Procesele care cer acces exclusiv sunt mai prioritare.
 Functiilor  care pot fi apelate de programele utilizator sunt:
int ExclOrSharedAccessInitialize(void); /*conectare la sistem */
void ExclOrSharedAccessEnd(void);    /* deconectare */
int OpenSharedAccess(void ); /* obtinere acces partajat */
int CloseSharedAccess(void ); /* obtinere acces partajat */
int OpenExclAccess(void );       /* obtinere acces exclusiv  */
int CloseExclAccess(void );      /* obtinere acces  exclusiv */
*****************************************************************/
/*****************************************************************
 Program client.
 Accesul este obtinut de la un server a carui localizare se
 determina cunoscind numele  de user sub care s-a deschis
 sesiunea la statia  serverului. Receptia mesajelor se face in mod
 asincron, intr-o functie EventServiceRoutine.

*****************************************************************/
#include <nxt.h>
#include <nit.h>
#include <string.h>
#include <mem.h>
#include <stdio.h>
#include <dos.h>

#define WAIT_FOR_MESSAGE 0
#define MAX_NO_PACKETS 10
#define DATA_LENGTH 15
#define USER_NAME_LENGTH 40
#define TRUE 1
#define FALSE 0
#define RECEIVE_ERROR 99
#define SEND_ERROR 98

int ExclOrSharedAccessInitialize(void);
void ExclOrSharedAccessEnd(void);
int OpenSharedAccess(void);
int CloseSharedAccess(void);
void far _loadds ReceiveESR(void);
int InitReceive(void);
void WaitResponse(void);
void ReceiveResponse(char *msg);
int SendRequest(char request, int cliConnectionNo, int
servConnectionNo);
void FindDestinationAddress( int *servConnectionNo );

WORD clientSocketRec=0x5000;
IPXHeader packetHeader;
ECB ecbR;
char packetData[DATA_LENGTH];
char queue[MAX_NO_PACKETS];
int cliConnNo, servConnNo;
int ind, code;

int ExclOrSharedAccessInitialize(void)
{
  int val;
  if ((code = IPXInitialize()) != 0) return 1;
  cliConnNo = (int)GetConnectionNumber();
  FindDestinationAddress(&servConnNo);
  if ((val = InitReceive()) != 0) return val;
  return 0;
}

void ExclOrSharedAccessEnd(void)
{
 IPXCancelEvent(&ecbR);
 IPXCloseSocket(clientSocketRec);
}

int OpenSharedAccess(void)
{
  char buffer;
  int val;
  if ((val = SendRequest('r',cliConnNo,servConnNo)) != 0)
          return val;
  WaitResponse();
  ReceiveResponse(&buffer);
  if(buffer != '@'){
          code = 0;
          return RECEIVE_ERROR;
  }
  return 0;
}

int CloseSharedAccess(void)
{
  char buffer;
  int val;
  if ((val = SendRequest('R',cliConnNo,servConnNo)) != 0)
          return val;
  WaitResponse();
  ReceiveResponse(&buffer);
  if(buffer != '@'){
          code = 0;
          return RECEIVE_ERROR;
  }
  return 0;
}

int OpenExclAccess(void)
{
  char buffer;
  int val;

  if ((val = SendRequest('w',cliConnNo,servConnNo)) != 0)
          return val;
  WaitResponse();
  ReceiveResponse(&buffer);
  if (buffer != '@'){
          code = 0;
          return RECEIVE_ERROR;
  }
  return 0;
}

int CloseExclAccess(void)
{
  char buffer;
  int val ;

  if ((val = SendRequest('W',cliConnNo,servConnNo)) != 0)
          return val;
  WaitResponse();
  ReceiveResponse(&buffer);
  if (buffer != '@'){
          code = 0;
          return RECEIVE_ERROR;
  }
  return 0;
}

int InitReceive(void)
{
if ((code = IPXOpenSocket((BYTE*)&clientSocketRec,0x00))!=0)
return 2;
 ind = 0;
 /* initializarea structurilor de date */
 ecbR.ESRAddress = ReceiveESR;
 ecbR.fragmentCount=2;
 ecbR.socketNumber=clientSocketRec;
 ecbR.fragmentDescriptor[0].address=&packetHeader;
 ecbR.fragmentDescriptor[0].size=sizeof(IPXHeader);
 ecbR.fragmentDescriptor[1].address=(BYTE *) packetData;
 ecbR.fragmentDescriptor[1].size=DATA_LENGTH;
```


BUPT

```
  /*deschiderea soclului si intrarea in "ascultare" de pachete*/
  IPXListenForPacket(&ecbR);
  IPXRelinquishControl();
  return 0;
}

void far _loadds ReceiveESR(void)
{
 if(ind <MAX_NO_PACKETS && ecbR.completionCode ==0){
   queue[ind] = packetData[0];
   ind++;
 }
 IPXListenForPacket(&ecbR);
}

void WaitResponse(void)
{
 int ind1;

 do{
       disable();
       ind1=ind;
       enable();
  }while (ind1 == WAIT_FOR_MESSAGE);
}

void ReceiveResponse(char *msg)
{
  int i;
  disable();
  if (ind>0) {
   *msg=queue[0];
   for (i=0; i<ind; i++) queue[i] = queue[i+1];
   ind--;
  }
  enable();
}

int SendRequest(char request, int cliConnNo, int servConnNo)
{
 BYTE serverNode[6];
 BYTE serverNetwork[4];
 WORD serverSocket = 0x5000,Socket;
 WORD clientSocketSnd = 0x6000;
 IPXHeader packetHeader;
 ECB ecb;
 char packetData[DATA_LENGTH];
 char buf[20];

 GetInternetAddress(servConnNo,(char *) serverNetwork,
                         (char *)serverNode,&Socket);
 movmem(serverNetwork,packetHeader.destination.network,4);
 movmem(serverNode,packetHeader.destination.node,6);
 movmem(&serverSocket,packetHeader.destination.socket,2);
 movmem(serverNode,ecb.immediateAddress,6);
 if((code =IPXOpenSocket((BYTE*)&clientSocketSnd,0x00))!=0)
 {
  IPXCancelEvent(&ecbR);
  return 3;
 }
 buf[0] = request;
 itoa(cliConnNo,&buf[1],10);
 buf[strlen(buf)+1] = '\0';
 strcpy(packetData,buf);
 ecb.ESRAddress = NULL;
 ecb.fragmentCount = 2;
 ecb.socketNumber = clientSocketSnd;
 ecb.fragmentDescriptor[0].address = &packetHeader;
 ecb.fragmentDescriptor[0].size = sizeof(IPXHeader);
 ecb.fragmentDescriptor[1].address = (BYTE *) packetData;
 ecb.fragmentDescriptor[1].size = strlen(packetData)+1;
 IPXSendPacket(&ecb);

 while(ecb.inUseFlag != 0) IPXRelinquishControl();
 if(ecb.completionCode != 0){
   IPXCancelEvent(&ecbR);
   IPXCloseSocket(clientSocketRec);
   code = 0;
   return SEND_ERROR;
 }
 IPXCloseSocket(clientSocketSnd);
 return 0;
}

void FindDestinationAddress( int *servConnNo )
{
        char name[USER_NAME_LENGTH] ;
        WORD numberOfConnections ;
        WORD connectionList[1] ;
        WORD maxConnections = 1 ;

   do{
         printf("\nLogin (user name for server): ") ;
         scanf("%s",name) ;

GetObjectConnectionNumbers(name,OT_USER,&numberOfCon
nections, connectionList, maxConnections);
         if(numberOfConnections == 0) printf("User name not
available. Try again!\n");
   }while (numberOfConnections == 0);
   *servConnNo = *connectionList ;
}

/*****************************************************************
Program server.
 Localizarea sa se determina cunoscind numele de user sub care
 s-a deschis sesiunea pe statia serverului
Receptia mesajelor se face in mod asincron, intr-o functie
 EventServiceRoutine.
 La receptia pachetelor, acestea sunt depuse intr-un buffer de
unde se preiau apoi pentru tratare.
*****************************************************************/
#include <nxt.h>
#include <nit.h>
#include <stdio.h>
#include <bios.h>
#include <string.h>
#include <mem.h>
#include <conio.h>
#include <dos.h>
#include <stdlib.h>

#define WAIT_FOR_MESSAGE 0
#define DATA_LENGTH 10
#define MAX_NO_PACKETS 10
#define MAX_NO_CONN 30
#define TRUE 1
#define FALSE 0

void Initialize(void);
void far _loadds ReceiveESR(void);
void InitReceive(void);
void WaitRequest(void);
void queueExtract(char *msg);
void ReceiveRequest(char *prequest);
void SendResponse(char ok[], int nrConn);
int GetPriProcess(int l[]);
int EmptyList(int l[]);
void OpenSharedAccess(void);
void CloseSharedAccess(void);
void OpenExclAccess(void);
void CloseExclAccess(void);

WORD serverSocketRec = 0x5000;
IPXHeader packetHeader;
ECB ecbR;
char packetData[DATA_LENGTH];
char *queue[MAX_NO_PACKETS];
int ind;
int SharedAccessNo,ExclAccessNo;
int inExclAccess;
int ShAccList[MAX_NO_CONN],ExclAccList[MAX_NO_CONN];
int cliConnNo;
```


BUPT

```
void main(void)
{
  int i;
  char request;
  Initialize();          /* initializarea comunicatiei */
  clrscr();
  SharedAccessNo = 0;          /* no de procese shared */
  ExclAccessNo = 0;            /* no de procese exlclusiv*/
  inExclAccess = FALSE;        /* acces liber */
  for(i=0;i<MAX_NO_CONN;i++){ /* liste vide */
    ShAccList[i] = 0;
    ExclAccList[i] = 0;
  }
  InitReceive();               /* initializare receptie */
 do{                           /* bucla de primire mesaje */
    WaitRequest();             /* asteptare cerere */
    ReceiveRequest(&request); /* receptie cerere */
    switch(request){           /* test tip cerere */
             case 'r':
               OpenSharedAccess();
               break;
             case 'R':
               CloseSharedAccess();
               break;
             case 'w':
               OpenExclAccess();
               break;
             case 'W':
               CloseExclAccess();
               break;
    }
  }while(1);
}

void Initialize(void)    /* initializare sistem comunicatie */
{
  if(IPXInitialize()!=0) {
    printf("\nIPX not installed. Abort!\n");
    exit(1);
  }
}

void InitReceive(void)   /* initializare receptie */
{
  int code,i;

 if ((code=IPXOpenSocket((BYTE*)&serverSocketRec,0x00))!=0)
 {
          printf("\nOpen socket error: %d. Abort!",code);
          exit(1);
 }
  ind=0;
  /* rezervarea cozii de pachete - pachetele se buffereaza */
  for (i=0; i<MAX_NO_PACKETS; i++)
    if((queue[i] = (char*)malloc(DATA_LENGTH))==NULL){
      printf("Memory allocation error. Abort!\n");
      exit(1);
    }
  /* initializarea structurilor de date */
  ecbR.ESRAddress=ReceiveESR;
  ecbR.fragmentCount=2;
  ecbR.socketNumber=serverSocketRec;
  ecbR.fragmentDescriptor[0].address=&packetHeader;
  ecbR.fragmentDescriptor[0].size=sizeof(IPXHeader);
  ecbR.fragmentDescriptor[1].address=(BYTE *) packetData;
  ecbR.fragmentDescriptor[1].size=DATA_LENGTH;

  /*deschiderea soclului si intrarea in "ascultare" de pachete*/
    IPXListenForPacket(&ecbR);
    IPXRelinquishControl();
 }

 void far _loadds ReceiveESR(void)
 /* rutina la nsata in cadrul intreruperii la primirea unui mesaj */
 {
   if (ind < MAX_NO_PACKETS && ecbR.completionCode == 0){
     strcpy(queue[ind],(char*)packetData);
     ind++;
   }
   IPXListenForPacket(&ecbR);
}

void WaitRequest(void)
/* functie pentru asteptarea sosirii unui mesaj */
{
 int ind1;

  do{
    disable();
    ind1=ind;
    enable();
  }while ((ind1 == WAIT_FOR_MESSAGE) && !bioskey(1));
  if (bioskey(1)){
          printf("\nServer terminated by user intervention!\n");
          IPXCancelEvent(&ecbR);
          IPXCloseSocket(serverSocketRec);
          exit(1);
  }
}

void queueExtract(char *msg)
/*extrage din coada de pachete urmatoarea cerere netratata*/
{
   int i;
   disable();
   if(ind>0){
     strcpy(msg,queue[0]);
     for(i=0;i<ind;i++)
      strcpy(queue[i],queue[i+1]);
     ind--;
    }
   enable();
}

void ReceiveRequest(char *prequest)
/*  preia  tipul urmatorului mesaj */
{
    char tmp[30];

    queueExtract(tmp);
    cliConnNo = atoi(&tmp[1]);
    printf("%d ",cliConnNo);
    *prequest = tmp[0];
}

void SendResponse(char ok[], int cliConnNo)
/* trimite un pachet la apelantul unei cereri*/
{
  WORD serverSocketSnd = 0x6000;
  WORD clientSocket = 0x5000,Socket;
  BYTE clientNode[6];
  BYTE clientNetwork[4];
  IPXHeader packetHeader;
  ECB ecb;
  char packetData[DATA_LENGTH];
  int code;
  strcpy(packetData, ok);
  GetInternetAddress(cliConnNo,(char *) clientNetwork,
                     (char *)clientNode,&Socket);
  movmem(clientNetwork,packetHeader.destination.network,4);
  movmem(clientNode,packetHeader.destination.node,6);
  movmem(&clientSocket,packetHeader.destination.socket,2);
  movmem(clientNode,ecb.immediateAddress,6);

  ecb.ESRAddress = NULL;
  ecb.fragmentCount = 2;
  ecb.socketNumber = serverSocketSnd;
  ecb.fragmentDescriptor[0].address = &packetHeader;
  ecb.fragmentDescriptor[0].size = sizeof(IPXHeader);
  ecb.fragmentDescriptor[1].address = (BYTE *) packetData;
  ecb.fragmentDescriptor[1].size = strlen(packetData)+1;
 if((code=IPXOpenSocket((BYTE*)&serverSocketSnd,0x00))==0)
  {
   IPXSendPacket(&ecb);
```


BUPT

```
  while(ecb.inUseFlag!=0) IPXRelinquishControl();
 }
 else {
  printf("Open socket error: %d. Abort!\n",code);
  IPXCancelEvent(&ecbR);
  IPXCloseSocket(serverSocketRec);
  exit (1);
 }
 IPXCloseSocket(serverSocketSnd);
}

int EmptyList(int list[])
/* testeaza daca  o lista  de identificatori este vida */
{
 int i;
 for (i=0; i<MAX_NO_CONN; i++)
   if (list[i] != 0)
             return 0;
 return 1;
}

int GetPriProcess(int list[])
/* preia identificatorul urmatorului  apelant
din lista de i dentificatori */

{
 int i;
 for (i=1; i<=MAX_NO_CONN; i++)
       if (list[i] != 0){
             list[i] = 0;
             return i;
       }
 return 0;
}

void OpenSharedAccess(void)
/* realizeaza obtinerea accesului pentru operatii partajabile asupra
 a bazei  da date */
{
 if (ExclAccessNo == 0){
   SharedAccessNo++;
   SendResponse("@",cliConnNo);
 }
 else
   ShAccList[cliConnNo] = cliConnNo;
}

void CloseSharedAccess(void)
/* realizeaza terminarea accesului i partajabil asupra  a bazei  da
date */
{
 SharedAccessNo--;
 SendResponse("@",cliConnNo);
 if((SharedAccessNo == 0) && (!EmptyList(ExclAccList))){
   cliConnNo = GetPriProcess(ExclAccList);
   inExclAccess = TRUE;
   SendResponse("@",cliConnNo);
 }
}


void OpenExclAccess(void)
/* realizeaza obtinerea accesului exclusiv asupra  a bazei  da date
*/
{
  ExclAccessNo++;
  if((SharedAccessNo == 0) && (inExclAccess == FALSE)){
   inExclAccess = TRUE;
   SendResponse("@",cliConnNo);
  }
  else
   ExclAccList[cliConnNo] = cliConnNo;
}

void CloseExclAccess(void)
/* realizeaza terminarea accesului  exclusiv asupra  a bazei  da
date */
{
 ExclAccessNo--;
 inExclAccess = FALSE;
 SendResponse("@",cliConnNo);
 if(!EmptyList(ExclAccList)){
    cliConnNo = GetPriProcess(ExclAccList);
    inExclAccess = TRUE;
    SendResponse("@",cliConnNo);
 }
 else
   while(!EmptyList(ShAccList)){
          cliConnNo = GetPriProcess(ShAccList);
          SharedAccessNo++;
          SendResponse("@",cliConnNo);
   }
}
```


BUPT

C5. Implementarea celei de-a treia soluţii bazate pe semafoare în reţele Unix şi Windows

```
/*****************************************************************/
Varianta Unix

 Suportul teoretic este prezentat in capitolul 5, la 5.1.1.
 Implementarea este descrisa la 5.1.3.2.
 Procesele care cer acces partajat sunt mai prioritare.
 Se folosesc semafoare distribuite a caror implementare a fost
 prezentata in anexa B7
 Functiile care pot fi apelate de programele utilizator sunt:

int ExclOrSharedAccessInitialize(void); /*conectare la sistem */
void ExclOrSharedAccessEnd(void);    /* deconectare */
void ExclOrSharedAccessIStart(void)  /* initializare semaf. */
void OpenSharedAccess(void );  /* obtinere acces partajat */
void CloseSharedAccess(void ); /* terminare acces partajat */
void OpenExclAccess(void );     /* obtinere acces exclusiv  */
void CloseExclAccess(void );    /* terminare acces  exclusiv */
*****************************************************************/
/*****************************************************************
Descrierea functiilor puse  la dispozitia programelor client

*****************************************************************/
/* definirea semafoarelor distribuite folosite */

int hPrioritySharedAcc= 3;
int hNoSharedAccess = 4;
int hWaitingExclAccess = 1;
int hExclusive = 2;
int NoSharedAccess;

void OpenSharedAccess(void){
SemaphoreDown( hExclusive );
 NoSharedAccess = SemaphoreUp( hNoSharedAccess );
 if (NoSharedAccess==1)SemaphoreDown( hPrioritySharedAcc);
 SemaphoreUp( hExclusive);
 }

void CloseSharedAccess(void){
   SemaphoreDown( hExclusive );
   NoSharedAccess = SemaphoreDown( hNoSharedAccess );
   if (NoSharedAccess==0) SemaphoreUp( hPrioritySharedAcc);
   SemaphoreUp( hExclusive );
}

void OpenExclAccess(void){
   SemaphoreDown( hWaitingExclAccess );
   SemaphoreDown( hPrioritySharedAcc);
}

void CloseExclAccess(void){
 SemaphoreUp( hPrioritySharedAcc);
 SemaphoreUp( hWaitingExclAccess );
}

int ExclOrSharedAccessInitialize(void){
 int errCode;
 if (errrCode = SemaphoreInitialize()) != 0)
    return errrCode;
  return 0;
}

void ExclOrSharedAccessIStart(void){
SemaphoreUp( hPrioritySharedAcc);
SemaphoreUp( hWaitingExclAccess );
SemaphoreUp( hExclusive );
}

void ExclOrSharedAccessEnd(void){
SemaphoreEnd();
}
```

```
/****************************************************************
                    Programul client

****************************************************************/
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <X11/Xaw/List.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Dialog.h>
#include <X11/Xaw/AsciiText.h>
#include "shexcl.h"

# define MAX_MES 50

typedef struct _ViewText{
        XawTextPosition    nrChText;
        unsigned int       nrRwText;
        XawTextPosition    chPos;
        unsigned int       rwPos;
        unsigned int       rowsNr;
        char               *text;
}ViewText;

Widget  widQuit,widInfo,widView, widPaned,
        widBoxMenu, widBoxView, widP, widV, widTopLevel,
        open_shared,open_excl,close_shared,close_excl,init;
void WriteMessage(Widget widDest,char *wString);
void AdjustDisplayPos(ViewText *display, int rowsNr);

void CallbackOpenShared
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackCloseShared
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackOpenWrite
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackCloseWrite
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackInit
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackInfo
        (Widget w,XtPointer client_data,XtPointer call_data);
void CallbackQuit
        (Widget w,XtPointer client_data,XtPointer call_data);

String SemList[MAX_SEM+1] = {NULL};

int        heightFont;

main(argc, argv)
int  argc;
char **argv;
{
 Dimension heightApp = 300,widthApp = 600,
           heightView, heightMenu;
  XtAppContext app_context;
  if ( ExclOrSharedAccessInitialize())
      exit(1);
  widTopLevel = XtVaAppInitialize(
        &app_context,
        "Main",
        NULL,0,
        &argc, argv,
        NULL,
        NULL);
   XtVaSetValues(widTopLevel,
          XtNx, 300,
          XtNy, 400,
          XtNwidth, widthApp,
          XtNheight, heightApp,
          NULL);
  widPaned = XtVaCreateManagedWidget(
      "mainPaned",
      panedWidgetClass,
```


BUPT

```
    widTopLevel,
    NULL) ;
widBoxMenu = XtVaCreateManagedWidget(
        "boxMenu",
        boxWidgetClass,
        widPaned,
        XtNorientation, XtorientHorizontal,
        NULL);
widInfo = XtVaCreateManagedWidget(
    "Info",
    commandWidgetClass,
    widBoxMenu,
    NULL);
open_shared= XtVaCreateManagedWidget(
        "OpenShared",
        commandWidgetClass,
        widBoxMenu,
        NULL);
close_shared= XtVaCreateManagedWidget(
        "CloseShared",
        commandWidgetClass,
        widBoxMenu,
        NULL);
open_excl = XtVaCreateManagedWidget(
        "OpenExcl",
        commandWidgetClass,
        widBoxMenu,
        NULL);
close_excl= XtVaCreateManagedWidget(
        "CloseExcl",
        commandWidgetClass,
        widBoxMenu,
        NULL);
init = XtVaCreateManagedWidget(
        "init",
        commandWidgetClass,
        widBoxMenu,
        NULL);
widQuit = XtVaCreateManagedWidget(
    "Quit",
    commandWidgetClass,
    widBoxMenu,
    NULL);
XtVaGetValues( widQuit, XtNheight, &heightMenu, NULL);
XawPanedSetMinMax( widBoxMenu, heightMenu+10,
                                heightMenu+10 );
widBoxView = XtVaCreateManagedWidget(
        "boxView",
        formWidgetClass,
        widPaned,
        NULL);
 widP = XtVaCreateManagedWidget(
        "widP",
        listWidgetClass,
        widBoxView,
        XtNlist, SemList,
        XtNwidth, 4,
        XtNheight, 0,
        XtNdefaultColumns,1,
        XtNforceColumns,True,
        NULL);
 widView = XtVaCreateManagedWidget(
        "Message",
        asciiTextWidgetClass,
        widBoxView,
        NULL);
 XtVaGetValues(widView,
        XtNheight,&heightFont,
        NULL);
 XtVaGetValues(widP,
        XtNheight,&heightView,
        NULL);
 XtVaSetValues(widView,
        XtNeditType, XawtextEdit,
        XtNscrollVertical,XawtextScrollAlways,
        XtNfromHoriz, widP,
        XtNheight, heightView,
        NULL);
widV = XtVaCreateManagedWidget(
        "widP",
        listWidgetClass,
        widBoxView,
        XtNlist, SemList,
        XtNwidth, 4,
        XtNheight, 0,
        XtNdefaultColumns,1,
        XtNforceColumns,True,
        XtNfromHoriz, widView,
        NULL);
XtAddCallback(open_shared,XtNcallback,
                        CallbackOpenShared,0);
XtAddCallback(close_shared,XtNcallback,
                        CallbackCloseShared,0);
XtAddCallback(open_excl,XtNcallback,CallbackOpenWrite,0);
XtAddCallback(close_excl,XtNcallback,CallbackCloseWrite,0);
XtAddCallback(init,XtNcallback,CallbackInit,0);
XtAddCallback(widInfo, XtNcallback, CallbackInfo, 0);
XtAddCallback(widQuit, XtNcallback, CallbackQuit, 0);


XtOverrideTranslations( widView,
                XtParseTranslationTable("<Key>:"));
XtRealizeWidget(widTopLevel);
XtAppMainLoop(app_context);
}

void CallbackInit(w,client_data,call_data)
Widget w;
XtPointer client_data,call_data;
{
    ExclOrSharedAccessStart();
}

void CallbackQuit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 ExclOrSharedAccessEnd();
 exit(0);
}

void CallbackInfo( w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
 char msg[MAX_MES], *adr;
  adr=GetMyAddress();
 sprintf(msg,"Server address  %s",adr);
 WriteMessage( widView, msg);
}

void CallbackOpenShared(w,client_data,call_data)
Widget w;
XtPointer client_data,call_data;
{
 OpenSharedAccess();
 WriteMessage(widView, "In shared access...");
}

void CallbackCloseShared(w,client_data,call_data)
Widget w;
XtPointer client_data,call_data;
{
 CloseSharedAccess();
 WriteMessage(widView, " Done shared access...");
 }

void CallbackOpenWrite(w,client_data,call_data)
Widget w;
XtPointer client_data,call_data;
{
 OpenExclAccess();
 WriteMessage(widView, "In exclusive access...");
}
void CallbackCloseWrite(w,client_data,call_data)
```


BUPT

```
Widget w;
XtPointer client_data,call_data;
{
 CloseExclAccess();
  WriteMessage(widView, " Done exclusive access...");
 }
void AdjustDisplayPos( display, rowsNr)
ViewText *display;
int   rowsNr;
{
XawTextPosition      iPos;
if( rowsNr != display->rowsNr ){
 display->rowsNr = rowsNr;
 for( iPos = display->nrChText; iPos && rowsNr; iPos-- )
  if(display->text[iPos] == '\n')
   rowsNr--;
  display->chPos = iPos ? iPos + 2 : 0L ;
 display->rwPos=display->nrRwText-( display->rowsNr -rowsNr );
}
  else
   if( display->rwPos + display->rowsNr <= display->nrRwText ){
     display->rwPos ++ ;     /* Scroll  o linie */
     for( ; display->text[display->chPos++] != '\n'; ) ;
   }
}

void WriteMessage(widDest,wString)
Widget     widDest;
char        * wString;
{
  static ViewText display           = { 0, 0, 0, 0, 0,NULL};
  char                    newLine[ 100 ];
  XawTextBlock       newText;
  Boolean               newDispPos;
  Dimension            heightWid;
  XawTextPosition    displayPos;
  strcpy( newLine, wString);
  strcat( newLine, "\n");
   XtVaGetValues( widDest,
              XtNdisplayPosition, &displayPos,
              XtNheight, &heightWid,
              XtNstring, &display.text,
              NULL);
   newText.firstPos = 0;
  newText.format = FMT8BIT;
  newText.ptr = newLine;
  newText.length = strlen(newLine);
   newDispPos = ( displayPos >= display.chPos );
   XawTextReplace( widDest, display.nrChText, display.nrChText,
                         &newText );
  display.nrChText += newText.length;
  display.nrRwText ++;
  AdjustDisplayPos( &display, heightWid/heightFont );
  if(newDispPos)
             XtVaSetValues( widDest, XtNdisplayPosition,
                                  display.chPos, NULL);
 }
/*******************************************************************/
 Varianta  Windows

  Suportul teoretic este prezentat in capitolul 5, la 5.1.1.
  Implementarea este descrisa la 5.1.3.2.
  Procesele care cer acces partajat sunt mai prioritare.
  Se folosesc semafoare distribuite a caror implementare a fost
  prezentata in anexa B7
  Functiile care pot fi apelate de programele utilizator sunt:

 int ExclOrSharedAccessInitialize(void);  /*conectare la sistem */
 void ExclOrSharedAccessEnd(void);     /* deconectare */
 void ExclOrSharedAccessIStart(void)  /* initializare semaf.*/
 void OpenSharedAccess(void );  /* obtinere acces partajat */
 void CloseSharedAccess(void ); /* terminare acces partajat */
 void OpenExclAccess(void );     /* obtinere acces exclusiv  */
 void CloseExclAccess(void );     /* terminare acces  exclusiv */

 ****************************************************************/
/*****************************************************************
Descrierea functiilor puse  la dispozitia programelor client

*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "semaforw.h"
#include "generic.h"

int hWaitingExclAccess = 1;         /* handlere semafoare */
int hExclusive = 2;
int hPrioritySharedAcc= 3;
int hNoSharedAccess = 4;
int NoSharedAccess;                   /* variabila partajata */

void OpenSharedAccess(void){
SemaphoreDown( hExclusive );
 NoSharedAccess = SemaphoreUp( hNoSharedAccess );
 if (NoSharedAccess==1)SemaphoreDown( hPrioritySharedAcc);
 SemaphoreUp( hExclusive);
 }

void CloseSharedAccess(void){
  SemaphoreDown( hExclusive );
  NoSharedAccess = SemaphoreDown( hNoSharedAccess );
  if (NoSharedAccess==0) SemaphoreUp( hPrioritySharedAcc);
  SemaphoreUp( hExclusive );
}

void OpenExclAccess(void){
  SemaphoreDown( hWaitingExclAccess );
  SemaphoreDown( hPrioritySharedAcc);
}

void CloseExclAccess(void){
 SemaphoreUp( hPrioritySharedAcc);
 SemaphoreUp( hWaitingExclAccess );
}

int ExclOrSharedAccessInitialize(void){
 int errCode;
 if ((errCode = SemaphoreInitialize()) != 0)
    return errCode;
 return 0;
}

void ExclOrSharedAccessIStart(void){
SemaphoreUp( hPrioritySharedAcc);
SemaphoreUp( hWaitingExclAccess );
SemaphoreUp( hExclusive );
}

void ExclOrSharedAccessEnd(void){
SemaphoreEnd();
}
/*****************************************************************
                     Programul client
 ****************************************************************/

LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg,
                   WPARAM wParam, LPARAM lParam );

HINSTANCE hInst;
LPCTSTR lpszAppName = "MyApp";
LPCTSTR lpszTitle   = "Shared-Exclusive";
int  APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE
           hPrevInstance,  LPTSTR lpCmdLine, int nCmdShow)
{
 MSG     msg;
 HWND     hWnd;
 WNDCLASS wc;
 WSADATA WSAData;
 UNREFERENCED_PARAMETER( lpCmdLine );
 UNREFERENCED_PARAMETER( hPrevInstance );
```


BUPT

```
if( GetVersion() & 0x80000000 && (GetVersion() & 0xFF) ==3){
 MessageBox( NULL, "This application cannot run on Windows
        3.1.\n" "This application will now terminate.", "WinApp",
        MB_OK I MB_ICONSTOP I MB_SETFOREGROUND );
    return( 1 );
 }
hInst = hInstance;
wc.style        = CS_HREDRAW I CS_VREDRAW;
wc.lpfnWndProc  = (WNDPROC)WndProc;
wc.cbClsExtra   = 0;
wc.cbWndExtra   = 0;
wc.hInstance    = hInstance;
 wc.hIcon       = LoadIcon (hInstance, lpszAppName);
 wc.hCursor     = LoadCursor(NULL, IDC_ARROW);
 wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
 wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
 wc.lpszClassName = lpszAppName;
 RegisterClass( &wc );
 hWnd = CreateWindow( lpszAppName,
                lpszTitle,
                 WS_OVERLAPPEDWINDOW,
                 CW_USEDEFAULT, 0,
                 CW_USEDEFAULT, 0,
                 NULL,
                 NULL,
                  hInstance,
                 NULL
                );
 if ( !hWnd )
     return( FALSE );
 ShowWindow( hWnd, nCmdShow );
 WSAStartup(MAKEWORD(1,1), &WSAData) ;
if (ExclOrSharedAccessInitialize()){
 MessageBox( NULL,  "This application will now terminate.",
 "WinApp", MB_OKIMB_ICONSTOPIMB_SETFOREGROUND);
   exit(1);
}
 while( GetMessage( &msg, NULL, 0, 0) )
       DispatchMessage( &msg );
 return( msg.wParam );
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg,
   WPARAM wParam, LPARAM lParam)
{
 HDC hdc;
 char str[10];
  int nr;
 switch (uMsg)
 {
 case WM_CREATE:
   return DefWindowProc( hWnd, uMsg, wParam, lParam );
   break;
 case WM_DESTROY:
   SemaphoreEnd();
   PostQuitMessage( 0 );
   break;
  case WM_COMMAND:
   switch ( LOWORD( wParam ) )
   {
    case ID_INIT:
      ExclOrSharedAccessIStart();
      break;
    case ID_SHARED:
      SetWindowText( hWnd, "Waiting for shared access" );
      OpenSharedAccess();
      hdc = GetDC(hWnd);
      TextOut(hdc,5,5,"          ",10);
      sprintf(str,"%d",nr);
      TextOut(hdc,5,5,str,strlen(str));
      ReleaseDC(hWnd,hdc);
      SetWindowText( hWnd, "In shared access..." );
      Sleep(10000);
      MessageBox( NULL, "Continue?",  "WinApp",
      MB_OKIMB_ICONQUESTIONIMB_SETFOREGROUND );
      hdc = GetDC(hWnd);
      TextOut(hdc,5,5,"          ",10);
      sprintf(str,"%s","End");
      TextOut(hdc,5,5,str,strlen(str));
       ReleaseDC(hWnd,hdc);
       CloseSharedAccess();
      hdc = GetDC(hWnd);
      TextOut(hdc,5,5,"          ",10);
      sprintf(str,"%d",nr);
      TextOut(hdc,5,5,str,strlen(str));
       ReleaseDC(hWnd,hdc);
        SetWindowText( hWnd, "Done  shared access" );
      break;
    case ID_EXCLUSIVE:
      SetWindowText( hWnd, "Waiting for exclusive access..." );
      OpenExclAccess();
      SetWindowText( hWnd, "In exclusive access..." );
      Sleep(10000);
      MessageBox( NULL, "Continue?", "WinApp",
      MB_OKIMB_ICONQUESTIONIMB_SETFOREGROUND);
      hdc = GetDC(hWnd);
      TextOut(hdc,5,5,"          ",10);
      sprintf(str,"%s","End");
      TextOut(hdc,5,5,str,strlen(str));
       ReleaseDC(hWnd,hdc);
      CloseExclAccess();
       SetWindowText( hWnd, "Done exclusive access" );
      break;
   case ID_EXIT:
      ExclOrSharedAccessEnd();
       DestroyWindow( hWnd );
      break;
   }
   break;
   default:
      return DefWindowProc( hWnd, uMsg, wParam, lParam );
   }
   return 0;
}
```


BUPT

ANEXA D
Implementarea sistemului DOSTP în rețele UNIX și WINDOWS

D1. Definirea clasei CObject

```
/* ***************************************************************
        Definitia clasei CObject in Unix                         *
*************************************************************** */
class CObject :public CDebug, public CError {
  public:
  CObjectManager(int debug=1, FILE *f =stdout); #constructor
  virtual int TreatRequest(PACKET *)=0;
  virtual int PrepareToCommit(void)=0;      // tratare faza prepare
  virtual int Commit(void)=0;               // tratare faza commit
  virtual int Abort(void)=0;                // tratare faza abort
  virtual int Interrupt(void)=0;            // nod. per. nu rasp.
  virtual int Continue(int, int)=0;         // recuperare A si C
  virtual int ObjectInit(void)=0;           // recuperare initiala
  virtual int Finish(void)=0;               // apel destructori
  virtual void SetTransNo(int, int)=0;      // setare transId si pri
  virtual GetInitData(char *)=0;            // obt. val obj pt depan.
  virtual void Startup(void)=0;             // initializari
  virtual void Cleanup(void)=0;             // dealocari finale
};
class CError {                              // pentru tratare erori
  public:
  int m_errCode=0;                          // codul erorii
  virtual void Error(int)=0;
};
class CDebug{       // pentru informatii la depanarea progr.
  public:
  int m_debug;           // flag, cu depanare sau fara
  FILE* m_debugFd;       // fisierul in care se depune informatia
  int m_pid;             // pid-ul
  void Debug(char*,...); // functia de scriere in fisier
  CDebug(int debug=1,FILE* debugFd=stdout) // constructor
   {m_pid=getpid();m_debug=debug;m_debugFd=debugFd;};
  void SetPid() {m_pid=getpid();}          // fixare pid
  void Init (int debug=1,FILE* debugFd=stdout) // initializari
            {m_debug=debug;m_debugFd=debugFd;}
  void Flush(){fflush(m_debugFd);}; // golire buffer
};
```

```
/* ***************************************************************
        Definitia clasei CObject in Windows                      *
*************************************************************** ********/
class CObject :public CDebug, public CError {
  public:
  CObjectManager(int debug=1, FILE *f =stdout); #constructor
  virtual int TreatRequest(PACKET *)=0;
  virtual int PrepareToCommit(void)=0;      // tratare faza prepare
  virtual int Commit(void)=0;               // tratare faza commit
  virtual int Abort(void)=0;                // tratare faza abort
  virtual int Interrupt(void)=0;            // nod. per. nu rasp.
  virtual int Continue(int, int)=0;         // recuperare A si C
  virtual int ObjectInit(void)=0;           // recuperare initiala
  virtual int Finish(void)=0;               // apel destructori
  virtual void SetTransNo(int, int)=0;      // setare transId si pri
  virtual GetInitData(char *)=0;            // obt. val obj pt depan.
  virtual void Startup(void)=0;             // initializari
  virtual void Cleanup(void)=0;             // dealocari finale
  virtual CObject * NewCopy(void)=0;        // singura fct.noua
};
class CError {                              // pentru tratare erori
  public:
  int m_errCode=0;                          // codul erorii
  virtual void Error(int)=0;
};
class CDebug{
  public:   int m_debug;
FILE* m_debugFd;
int m_pid;
void Debug(char*,...);
CDebug(int debug=1,FILE* debugFd=stdout) // constructor
        m_pid=GetCurrentThreadId();
        _debug=debug; m_debugFd=debugFd;};
void SetPid(){m_pid=GetCurrentThreadId();}
void Init(int debug=1,FILE* debugFd=stdout)
        m_debug=debug;m_debugFd=debugFd;}
void Flush(){fflush(m_debugFd);};
};
```

D2. Definirea clasei CVector

```
/* ***************************************************************
        Definitia clasei CVector  in Unix                        *
*************************************************************** */
class Cvector : public CObject {
 // membri privati:
  char m_serverName[MAX_OBJ_LEN];
  char m_dataFile[MAX_PATH], m_statFile[MAX_PATH];
  char m_logFilePC[MAX_PATH], m_logFileAC[MAX_PATH;
  char m_tsFile[MAX_PATH],       // numele fisierelor
  FILE * m_lfPC,*m_lfAC,*m_tf;   //LogFile,TsFile
  int m_df, m_sf;  // descriptori pt dataFile
  long m_logPos;       // poz articolelor in fisierul PCLOG
  int    *m_pData ; // pointer la zona de mem. cu fis de date
  STATINF *m_pTrStatus ; // pointer la mem. cu inf de stare
  int m_semObId;     // semafor pt. acc. excl. la inf. de stare
  int m_semTs;       // semafoare pt. acc. excl. la fis. log
  int m_transId;     // identificatorul tranzactiei
  int m_pri;         // prioritatea tranzactiei
  int m_transIdx;    // indexul in vectorul de tranz al acestei tranz
  WR m_vWPos[MAX_OB]; // perechi index,value pt. write
  int m_wIdx;        // urm intrare ibera din m_vWPos
  int m_vRPos[MAX_OB];  // pozitiile citite
  int m_rIdx;        // urm intrare libera din m_vRPos
  int m_status;      //starea tranzactiei :
                     // OK_PC, FAIL_PC, WORKING_PC
 // metode private
   void ClearLogPC(void);         // rescrie PCLOG
   void InitialRec(void);         // recupereaza tranz. din ACLOG
   int Read(VEC_REQUEST *p); // citire o pozitie
   int Write(VEC _REQUEST*p); // scriere o pozitie
   int AllocInVectTz(void) ;      // aloca o intr. dn vect. cu tr. active
   int ReleaseInVectTz(int) ;     // elibereaza o intr. din vect. de tr.
   void RecordOldTz(void);        // act. tz. old citind PCLOG
   void  OldTzStart( int index);  // act. starea unei tz. old la WORK
   void  OldTzDone( int index);   // act. starea unei tz. old la DONE
   short OldTzStat(int idTranz); // return. starea unei tz old
   void InOldTz(void);            // obt. acc. excl. cu  m_semTs
   void OutOldTz(void);           // elib. acc. excl. cu  m_semTs
   long LogPcAddr(int idTz, int *index,int *prior); // ret. adr. din PC
   int InTF(void);                // ret. daca tz. a fost terminata
   int InVectTz(void);            // ret. daca tz este in lucru
   void ExclAcces(void);          // obt. acc. excl. cu  m_semObId
   void ReleaseAcces(void);       // elib. acc. excl. cu  m_semObId
   void SemWait(int);             // astept. la  sem. param.
   void SemSignal(int);           // Up la semaf. param.
   off_t DimFile(int ) ;          //  determ. dim. fis. de date
   void DCVector(void);           // rol de destructor
   void PrintStatus(void);        // inf. pt. depanare

   public:
   INCLUDE_SISTEM (CVector)
   CVector (char *name, char* dataFile, char* logFilePC,char*
                      logFileAC,char * tzFile, char* statFile );
   CVector(){};
```


BUPT

```
 virtual int TreatRequest(PACKET *); // tratarea unui pckt SHM
 virtual int PrepareToCommit(void ); // faza prepare
 virtual int Commit(void);          // terminare anormal tr.
 virtual int Abort(void);           // terminare anormal tr.
 virtual int Interrupt(void);       // pentru cind TM nu mai rasp.
 int Continue(int tz,int op);       // recuperarea tz. respective
 int ObjectInit();                  // recupereaza si rescrie PCLOG
 virtual void SetTransNo(int transNo,int pri)
    {m_transId=transNo;m_pri=pri;} // setare transId, pri
 virtual void Startup(void);        // initializari
 virtual void Cleanup(void);        // dealocari
 virtual void Finish(void);         // dealocari finale
 void GetBaseData(char *);          // afiseaza pt. depanare val. ob.
 void Error(int){};                 // afisarea erorilor
};

/* ***************************************************************
          Definitia clasei CVector in Windows                    *
************************************************************ ********* */
class Cvector : public CObject {
 // membri privati
 char m_serverName[MAX_OBJ_LEN], m_statFile[MAX_PATH];
 char m_logFilePC[MAX_PATH], m_logFileAC[MAX_PATH];
 char m_tsFile[MAX_PATH],       // numele fisierelor
 FILE * m_lfPC,*m_lfAC,*m_tf;   // LogFile,TsFile
 long m_logPos;      // poz articolelor in fisierul PCLOG
 HANDLE m_df;                 //handle pt dataFile
 HANDLE m_hData; // handle al zonei de mem partajata pt ob
 int   *m_pData ;     // pointer la zona de mem. cu fis de date
 STATINF *m_pTrStatus ; // pointer la zona de mem. cu inf de
stare    HANDLE m_semObId; //f pt acc. excl. inf de stare
 HANDLE m_semTz; // pt acc. excl. la inf. de stare pt . tz old
 int m_transId;        // identificatorul tranzactiei
 int m_pri;            // prioritatea tranzactiei
 int m_transIdx;       // indexul in vectorul de tranz al acestei tranz
 WR m_vWPos[MAX_OB]; // perechi index,value pt. write
 int m_wIdx;           // urm intrare ibera din m_vWPos
 int m_vRPos[MAX_OB]; // pozitiile citite
 int m_rIdx;           // urm intrare libera din m_vRPos
 int m_status;         //starea tranzactiei :
                       // OK_PC, FAIL_PC, WORKING_PC
 // metode private
 void ClearLogPC(void);         // rescrie PCLOG
 void InitialRec(void);         // recupereaza tranz. din ACLOG
 int Read(VEC_REQUEST *p); // citire o pozitie
 int Write(VEC _REQUEST*p); // scriere o pozitie
 int AllocInVectTz(void) ;      // aloca o intr. dn vect. cu tr. active
 int ReleaseInVectTz(int) ;     // elibereaza o intr. din vect. de tr.
 void RecordOldTz(void);        // act. tz. old citind PCLOG
 void OldTzStart( int index);   // act. starea unei tz. old la WORK
 void OldTzDone( int index);    // act. starea unei tz. old la DONE
 short OldTzStat(int idTranz);  // return. starea unei tz old
 void InOldTz(void);            // obt. acc. excl. cu m_semTs
 void OutOldTz(void);           // elib. acc. excl. cu m_semTs
 long LogPcAddr(int idTz, int *index,int *prior); // ret. adr. din PC
 int InTF(void);                // ret. daca tz. a fost terminata
 int InVectTz(void);            // ret. daca tz este in lucru
 void ExclAcces(void);          // obt. acc. excl. cu m_semObId
 void ReleaseAcces(void);       // elib. acc. excl. cu m_semObId
 void SemWait(HANDLE);          // astept. la sem. param.
 void SemSignal(HANDLE);        // Up la semaf. param.
 off_t DimFile(int ) ;          // determ. dim. fis. de date
 void DCVector(void);           // rol de destructor la Quit
 void PrintStatus(void);        // inf. pt. depanare

public:
 INCLUDE_SISTEM (CVector)
 CVector (char *name, char* dataFile, char* logFilePC,char*
     logFileAC,char * tzFile, char* statFile );   // constructor
 CVector(){};
 CBase * NewCopy(void); // creaza dinamic o alta instanta a ob.
  virtual int TreatRequest(PACKET *); // tratarea unui pckt SHM
  virtual int PrepareToCommit(void); // faza prepare
  virtual int Commit(void);        // terminare anormal tr.
  virtual int Abort(void);         // terminare anormala tr.
  virtual int Interrupt(void);     // pentru cind TM nu mai rasp.
 int Continue(int tz,int op);      // recuperarea tz. respective
 int ObjectInit(void);             // recupereaza si rescrie PCLOG
 virtual void SetTransNo(int transNo,int pri)
    {m_transId=transNo;m_pri=pri;} // setare transId, pri
 virtual void Startup(void); // initializari inainte de rec. pckt SHM
 virtual void Cleanup(void); // dealocari dupa abort, commit SHM
 virtual void Finish(void);       // dealocari finale
 void GetBaseData(char * str); // ret. valorile ob. pt. depanare
 void Error(int){};             // afisarea erorilor
};
```

D3. Definirea clasei CObjectManager

```
/* ***************************************************************
          Definitia clasei CObjectManager in Unix                *
**************************************************************** */
class CObjectManager: public CError,public CDebug{
  // membri privati
   Widget m_label,m_quit,m_box;
  XtInputId m_sockId;
  int m_aut; // flag care indica daca ob. s-a conectat sau nu
  int m_finished; // folosit pt. a evita dubla dealocare:quit si ies.
  int m_transNo;     // numarul asociat tranzactiei
  CClientList m_clientList; // lista tranzactiilor ce acceseaza ob.
  char pathName[MAX_PATH] // nume pt. dir. curent
  struct sockaddr_un m_serv_addr,m_cli_addr; // adrese
  int m_clilen;            // dim. adresei
  int m_chan_U_OBJ;    // canalul static (beg. abort, com)
  int m_chanf_CS;  // pentru comunic. cu CS
  BT_OBJECT m_mes; // cererea curenta SNextRequest*/

 public:
  CObject * m_pObject; // pointer la ob. gestionat
  char m_objName[MAX_OBJ_LEN];
  void XInit(XtAppContext context,Widget pane);
  int Init (XtAppContext context, Widget pane, char* name, int
                   debug=1,FILE * debugFd=stdout);
  int ConnectObjectToSystem(void); // fctia pt. conectarea ob.
  int CreateSock(void);        // crearea canalului propriu
  int RecordObj(void);         // inregistreaza ob. la CM
  void DisconnectObjectFromSystem(void); // deconectare sist.
  void TransactionList(void);  // afisarea tranzactiilor active
  int AddClient(char * name,int transId,int pri); // adauga o tr.
  int RemoveClient(char* name); // elimina un client name
  int RemoveClient(int transId); // elimina un client transId
  int SNextRequest(void);        // trateaza urm.cerere din socket
  void Error(int);                  // tratarea erorilor
  int GetTzNo(){return m_tzNo;} // ret. nr tranzactie
  void DCObjectManager(void);   // simuleaza destructor
  SigFunc* signal(int signo,SigFunc* func);
   friend  void sig_chld(int signo);
   friend void Quit(Widget w,XtPointer client_data, XtPointer
                                        call_data);
   friend  int SNextRequest(XtPointer,int * ,XtInputId *); //
           trateaza urmatoarea cerere care vine in canalul static
};

/* ***************************************************************
          Definitia clasei CObjectManager in Windows             *
*************************************************************** */
class CObjectManager: public CError,public CDebug{
  // membri privati
  int m_aut; // flag care indica daca ob. s-a conectat sau nu
  int m_finished; // folosit pt. a evita dubla alocare: quit si ies.
  CClientList m_clientList; // lista tranzactiilor ce acceseaza ob.
  HANDLE m_listMutex;    // semaf. pentru acc. excl.
  SOCKADDR_IN m_serv_addr,m_cli_addr; // adrese Inet
  int m_clilen;             // dim. adresei
  SOCKET m_chan_K_OBJ; // canalul static (beg. abort, com)
  SOCKET m_chan_K_CS;  // pentru comunic. cu CS
  SOCKET m_newsockfd; // canal dinamic

  u_short m_myPort;    // portul asociat managerului de ob.
  HANDLE m_copyMutex,m_gCopyMutex; // pt. copierea inf.
  BT_OBJECT m_mes; // cererea curenta SNextRequest*/
```


BUPT

```
public:
  CObject * m_pObject;  // pointer la ob. gestionat
  char m_objName[MAX_OBJ_LEN];
  int  Init (CObject *pServer,char* name,int port,HANDLE
     gCopyMutex, int debug, FILE* debugFd);  // initializari
  int ConnectObjectToSystem(void); // fctia pt. conectarea ob.
  int   CreateSock(void);          // crearea canalului propriu
  int   RecordObj(void);           // inregistreaza ob. la CM
  void DisconnectObjectFromSystem(void); // deconectare sist.
  void TransactionList(void);   // afisarea tranzactiilor active
  int   AddClient(char * name,int transId,int pri); // adauga o tr.
  int   RemoveClient(char* name); // elimina un client name
  int   RemoveClient(int  transId); // elimina un  client transId
  int  SHMServer(void); // creaza  un fir pt. comun. SHM
  int  RecoveryServer(void);  // creaza  un fir pt. recuperare
  int SNextRequest(void)      // trateaza urm.cerere  din socket
  BT_OBJECT  GetMessage(void); // ret. mes. primi t in socket
  HANDLE GetCopyMutex(void);    // ret. handle-ul pt  CopyM
  HANDLE GetGCopyMutex(void);  // ret. handle-ul pt  gCopyM
  SOCKET GetNewSockFd(void);   // ret. m_newsockfd
  CObject * GetPObject(void);    // ret. m_pObject
  void  Error(int);              // tratarea erorilor
  void DCObjectManager(void);  // simuleaza destructor
};
```

D4. Definirea clasei CDispatcher

```
/* ****************************************************************
                Definitia clasei CDispatcher in Unix
****************************************************************/
class  CDispatcher: public CError , public CDebug{
  // membri privati
  int m_transId;         // identificatorul tranzactiei
  int m_pri;             // prioritatea tranzactiei
  char  m_path[MAX_PATH];          // cale socket unix
  struct sockaddr_un m_serv_addr;   // adrese
  CObject *m_pObject;  // pointer la obiect ;
  BT_OBJECT  m_mes; // cererea curenta  SNextRequest
  PACKET m_pcket;   // pckt primit in SHM
  int m_semId;            // id. pt. setul de semaf. pt. SHM
  int m_semNr;            //  nr. semaf.  pt. obj. din set
  int m_shmId;            // id. pt. mem. partajata
  SHM * m_pShm;           // pointer la mem. partajata
  int m_queueNo;         // nr. cozii rezervate in  tranz. pt. ob.
  int m_queueIdx ;        //  indicele pkt-ului urm. din coada
  int m_waiting;          // flag pentru bucla de asteptare

public:
  int Init(int debug=1,FILE *debugFd=stdout)  // initializari
  {m_queueIdx =0;m_waiting=1;CDebug::Init(debug,debugFd);}
  int WrongTzNo(void); // verifica nr. tz. din pckt cu m_transId
  void SetPacketErr(int code){m_pckt.text.errCode=code ;}
  void  MainJob(void);  // bucla principala a serverului
  int  Attach(void);    // atasare la SHM
  void  Dettach(void){ shmdt((char * )m_pShm);}
  int  ResolveTrans(BT_OBJECT *,char* ); // gestioneaza tr.
  int  WaitForAMessage(void); // asteptarea unui pckt in SHM
  void TreatMessage(void);      // tratare mesaje
  void FirstRequest(void);      // tratare prima cerere la ob
  void Request(void) ;          // tratare cerere
  void PrepareToCommitTr(void) ; // tratare prepare
  void CommitTr(void) ;          // tratare commit
  int AbortTr(void);             // tratare abort
  int  InterruptTr(void);        // tratare interrupt
  int SendPckt(void);            // emisie pckt la CS prin SHM
  int Up_S(int );                // Up la semaf.param
  int Down_S(int );              // Down la semaf.param
  void FreeSlot(int);            // Elibereaza un slot din SHM
  int GetSlot(void);             // Aloca un slot in SHM
  void SetClasDeriv (CObject * athis){m_pServer=athis;}
  void SetTransNo(int);          // stabileste nr. tranzactiei
  void FinishTr(void){m_pObject->Finish();}
   void GetBaseData(char* p){m_pObject->GetBaseData(p);}
  void RecoveryTr(BT_OBJECT* ,int ,char *); // recupere tr.
  void  Error(int code);         // traterea erorilor
};
```

```
/* ****************************************************************
                Definitia clasei CDispatcher in Windows
****************************************************************/
class  CDispatcher: public CError , public CDebug{
  // membri privati
  int m_transId;         // identificatorul tranzactiei
  int m_pri;             // prioritatea tranzactiei
  CObject *m_pObject;  // pointer la obiect ;
  BT_OBJECT  m_mes; // cererea curenta  SNextRequest
  PACKET m_pcket;   // pckt primit in SHM
  SHM * m_pShm;         // pointer la mem. partajata
  HANDLE m_sShm, m_sCs, m_sOb, m_sSlot, m_hShm
  int m_queueNo;         // nr. cozii rezervate in  tranz. pt. ob.
  int m_queueIdx ;        //  indicele pkt-ului urm. din coada
  int m_waiting;          // flag pentru bucla de asteptare
  CObjectManager   * m_pManager; // pointer la manager

public:
  int Init(int debug=1,FILE *debugFd=stdout)  // initializari
  {m_queueIdx =0;m_waiting=1;CDebug::Init(debug,debugFd);}
  int WrongTzNo(void); // verifica nr. tz. din pckt cu m_transId
  void SetPacketErr(int code){m_pckt.text.errCode=code ;}
  void  MainJob(void);  // bucla principala a serverului
  int  Attach(void);    // atasare la SHM
  void  Dettach(void){
     UnmapViewOfFile( (LPSTR) m_pShm );
      CloseHandle(m_sShm);
      CloseHandle(m_sMC);
      CloseHandle(m_sOb);
      CloseHandle(m_sSlot);
      CloseHandle(m_hShm);
   }
  int  ResolveTrans(BT_OBJECT *,char* ); // gestioneaza tr.
  int  WaitForAMessage(void); // asteptarea unui pckt in SHM
  void TreatMessage(void);      // tratare mesaje
  void FirstRequest(void);      // tratare prima cerere la ob
  void Request(void) ;          // tratare cerere
  void PrepareToCommitTr(void) ; // tratare prepare
  void CommitTr(void) ;          // tratare commit
  int AbortTr(void);             // tratare abort
  int  InterruptTr(void);        // tratare interrupt
  int SendPckt(void);            // emisie pckt la CS prin SHM
  int Up_S(int );                // Up la semaf.param
  int Down_S(int );              // Down la semaf.param
  void FreeSlot(int);            // Elibereaza un slot din SHM
  int GetSlot(void);             // Aloca un slot in SHM
  void SetClasDeriv (CObject * athis){m_pServer=athis;}
  void SetTransNo(int);          // stabileste nr. tranzactiei
  void FinishTr(void){m_pObject->Finish();}
   void GetBaseData(char* p){m_pObject->GetBaseData(p);}
  void RecoveryTr(BT_OBJECT* ,int ,char *); // recupere tr.
  void  Error(int code);         // traterea erorilor
  void  SendMessageToParent(int);
  void SetPMng(CObjectManager* pMng){m_pManger=pMng;}
};
```


BUPT

D5. Definirea macroinstrucțiunilor pentru conectarea automata a obiectelor in sistemul DOSTP

```
/*****************************************************************
            Definirea macroinstruciunilor in Unix                *
*****************************************************************/
#define INCLUDE_SISTEM (className)\
   CObjectManager * m_master;\
   void friend Init(className* p,XtAppContext,Widget );

#define  DEFINE_SISTEM(className)\
  void Init(className* s,XtAppContext context,Widget pane)\
  { CObjectManager*  master=new(CObjectManager);\
    s->m_master=master;\
    master->m_pObject=s;\
    master->Init (context, pane, s->m_serverName, s->m_debug,\
                                   s->m_debugFd);\
    s->Debug(" Master executed Init ");\
  }

#define  START_SISTEM(server,context,pane) \
 Init(server,context,pane)

#define STOP_SISTEM(obj) \
 obj->m_master->DisconnectObjectFromSystem(void)

/*****************************************************************
            Definirea macroinstruciunilor in Windows             *
*****************************************************************/

#define INCLUDE_SYSTEM(className)\
    u_short  m_port;\
    HANDLE  m_copyMutex;\
    void friend Init(className* pObject, u_short  port);\
    void friend ThreadMaster(className *);

#define DEFINE_SYSTEM(className)\
  void Init(className* pObject,u_short port)\
  { pObject->m_port=port;\
    char name[MAX_OBJ_LEN];\
    strcpy(name,pObject->m_serverName);\
   strcat(name,"copyMutex");\
    pObject->m_copyMutex=CreateSemaphore(NULL,0,1,name);\
    if (!pObject->m_copyMutex)\
           return ;}\
   HANDLE h;\
   DWORD idT;\
   h=CreateThread((NULL,0,(LPTHREAD_START_ROUTINE) \
                        ThreadMaster, pObject,0,&idT);\
   if (!h) {MsgBox("Create Thread error \n");}\
 }\
 void ThreadMaster(className *pObject)\
 {\
   CObjectManager  master;\
   master.Init(pObject,pObject->m_serverName,\
      pObject->m_port,pObject->m_copyMutex,\
      pObject->m_debug, pObject->m_debugFd);\
   HANDLE h; \
   DWORD idT; \
   h=CreateThread (NULL,0,(LPTHREAD_START_ROUTINE)\
            ThreadDisplay,&master,0,&idT);\
   if (!h) {MsgBox(" Create Thread error n");\
   while(TRUE)\
      master.SNextRequest();\
 }

#define START_SYSTEM(pObject,port) \
 Init(pObject,port)

#define OUT_SYSTEM(obj) \
 obj->m_master.DisconnectObjectFromSystem(void)
```


BUPT

ANEXA E

Comparaţie între performanţele tranzacţiilor distribuite si cele ale tranzacţiilor locale în sistemul DOSTP

Această anexă prezintă comparativ performanţele tranzacţiilor distribuite faţă de cele locale. Pentru testare s-au folosit 3 manageri de obiecte, într-o prima variantă fiecare pe cîte o staţie şi într-o altă variantă toţi local, pe aceeaşi staţie. S-a folosit acelaşi program client insă utilizînd 1,2,3,4,5 tranzacţii care implică cele 3 obiecte. S-au afişat următoarele valori (vezi tabelul următor): timpul total cît a durat execuţia programului client, timpul utilizator în care s-a executat programul client şi timpul sistem în care s-a executat programul client. Timpul a fost reprezentat in tick-uri de 1/100 sec.
Rezultatele sunt descrise în tabelul următor.

1) Tranzacţii asincrone locale

	(1)	(2)	(3)	(4)	(5)
Timp total	2,192	3,372	4,347	5,247	6,582
Timp utilizator	1	2	2	4	4
Timp sistem	4	6	10	14	16

2) Tranzacţii sincrone locale

	(1)	(2)	(3)	(4)	(5)
Timp total	2,258	3,439	4,713	5,450	6,626
Timp utilizator	1	2	2	3	3
Timp sistem	4	6	12	15	18

3) Tranzacţii asincrone distribuite

	(1)	(2)	(3)	(4)	(5)
Timp total	952	1,369	1,896	2,333	2,797
Timp utilizator	1	2	2	2	2
Timp sistem	4	8	11	13	16

4) Tranzacţii sincrone distribuite

	(1)	(2)	(3)	(4)	(5)
Timp total	2,557	4,574	4,980	6,788	7,243
Timp utilizator	2	2	2	3	3
Timp sistem	5	6	13	17	19

BUPT

Se observă că:
(1) Timpul necesar realizării tranzacţiilor distribuite sincrone este cel mai mare;
(2) Timpul necesar realizării tranzacţiilor distribuite asincrone este cel mai mic;
(3)Timpul necesar realizării tranzacţiilor care se execută local este aproximativ de 2.5 (numărul de obiecte implicate în tranzacţie este 3 dar intervine şi timpul necesar protocolului two phase commit) ori mai mare decît timpul necesar realizării aceloraşi tranzacţii pe alte staţii.
Aceste concluzii sunt reprezentate în figura următoare.

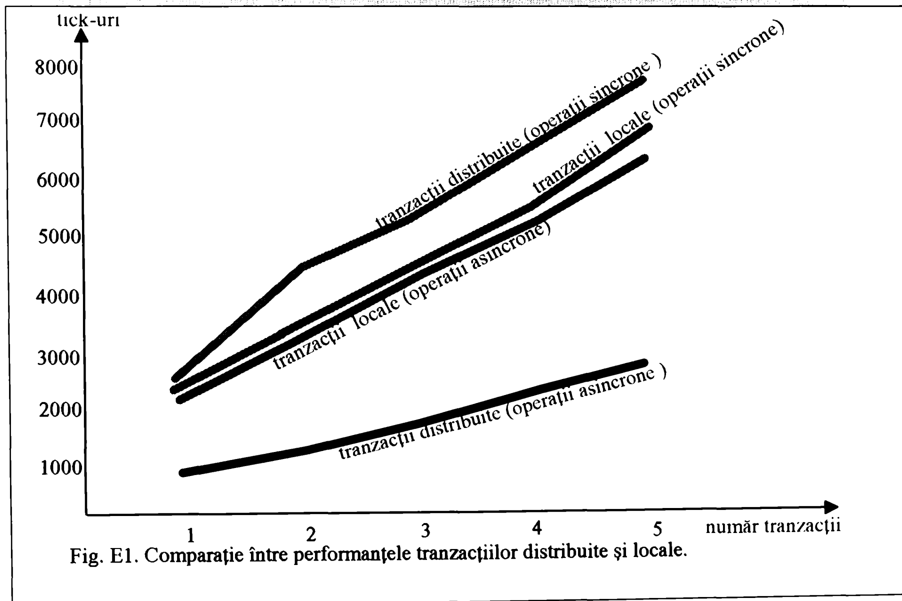


Fig. E1. Comparaţie între performanţele tranzacţiilor distribuite şi locale.

BUPT

ANEXA F

Comparaţie între performanţele variantelor distribuite şi semidistribuite ale sistemului TGC

Această anexă prezintă rezultatele obţinute în urma testării sistemului TGC. Testele s-au realizat prin execuţia unor programe client care trimit în buclă continuă mesaje către un program client receptor. Pentru comparaţie, programele emiţătoare s-au executat în mai multe variante: pe aceeaşi staţie cu programul emiţator (1, 2 , 3 şi 4 programe emiţătoare) şi, la distanţă, pe 1, 2, 3 sau 4 staţii; tot pentru comparaţie, s-au înregistrat rezultatele pentru trimiterea a 50, 100, 200, 300, 400, 500 şi 1000 mesaje. S-au afişat următoarele valori (vezi tabelul următor):
- pentru programele client emiţătoare: (1) numărul de mesaje trimise; (2) numărul de mesaje retrimise în urma unei erori de tipul " server de comunicaţie ocupat"; (3)timpul total cît a durat execuţia; (4) timpul utilizator în care s-a executat programul; (5) timpul sistem în care s-a executat programul;
- pentru programul receptor: (6) numărul de mesaje recepţionate; (7) numărul de mesaje pierdute; (8) timpul total cît a durat execuţia; (9) timpul utilizator în care s-a executat programul; (10) timpul sistem în care s-a executat programul. Timpul a fost reprezentat in tick-uri de 1/100 sec.
Rezultatele sunt descrise în tabelele următoare.

1. Varianta distribuită cu transmisie broadcast

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Un client la distanţă	50	0	31	4	3	50	0	31	0	1
	100	0	62	1	2	100	0	61	2	3
	200	0	128	1	7	200	0	127	5	3
	300	0	200	8	9	300	0	198	3	7
	400	0	301	12	10	400	0	308	8	9
	500	0	391	8	9	500	0	389	11	9
	1,000	0	949	22	29	1,000	0	948	15	9
2 clienti la distanţă	50	0	36	2	2	100	0	176	3	0
	100	0	81	7	2	200	0	304	5	7
	200	0	209	5	5	400	0	518	7	14
	300	0	483	6	5	600	0	862	10	9
	400	0	698	8	10	800	0	1,245	16	23
	500	0	1,119	14	11	1,000	0	1,610	27	27
	1,000	0	2,712	31	28	2,000	0	2,900	46	60
	50	0	38	1	1					
	100	0	89	1	1					
	200	0	247	5	8					
	300	0	597	6	8					
	400	0	848	7	9					
	500	0	1,309	7	10					
	1,000	0	2,823	27	27					
3 clienti la distanta	50	0	59	1	1	150	0	118	3	4
	100	0	225	1	3	300	0	276	4	3
	200	0	749	1	4	600	0	796	13	5
	300	0	769	10	8	900	0	1,640	18	21
	400	0	844	12	17	1,200	0	1,822	26	36
	500	0	1,449	16	17	1,500	0	2,460	23	22
	1,000	0	3,705	28	27	3,000	0	6,236	47	70
	50	0	62	0	4					
	100	0	228	1	2					
	200	0	738	1	3					
	300	1,467	1,325	31	41					
	400	977	1,599	29	26					
	500	1,287	2,223	21	34					
	1,000	2,749	5,526	80	93					

BUPT

	50	0	64	1	1					
	100	0	166	4	4					
	200	0	682	4	12					
	300	1,528	1,327	36	50					
	400	0	1,062	30	30					
	500	1,315	3,261	37	37					
	1,000	2,758	5,673	69	68					
4 clienti la distanță	50	0	25	2	0	200	0	224	5	4
	100	0	52	2	2	400	0	328	6	11
	200	58	258	8	4	800	0	745	13	23
	300	578	838	19	23	1,200	0	1,763	28	18
	400	1,517	2,901	37	35	1,600	0	4,692	44	40
	500	5,681	4,568	112	168	2,000	0	5,601	43	52
	1,000	18,243	14,356	349	510	4,000	0	14,976	105	192
	50	0	25	4	0					
	100	0	68	1	2					
	200	77	348	8	7					
	300	583	1,417	21	26					
	400	1,578	4,360	44	45					
	500	0	3,659	22	25					
	1,000	13,002	10,854	262	362					
	50	0	38	1	0					
	100	0	90	3	2					
	200	0	282	3	1					
	300	877	1,367	32	34					
	400	0	3,204	7	11					
	500	5,660	5,244	121	110					
	1,000	16,044	14,672	235	498					
	50	0	33	0	1					
	100	0	87	2	1					
	200	0	360	4	4					
	300	807	1,439	18	38					
	400	0	3,466	10	9					
	500	6,566	5,246	131	190					
	1,000	19,595	14,672	325	565					
Un client local	50	0	22	1	1	50	0	21	1	1
	100	0	50	3	4	100	0	50	1	2
	200	0	106	6	0	200	0	105	1	5
	300	0	189	5	6	300	0	189	5	8
	400	0	250	13	11	400	0	251	8	8
	500	0	336	9	15	500	0	336	9	4
	1,000	0	717	12	17	1,000	0	717	21	26
2 clienti locali	50	0	22	1	1	100	0	140	1	3
	100	0	53	1	2	200	0	157	2	6
	200	0	115	2	0	400	0	362	10	8
	300	0	206	5	7	600	0	429	7	14
	400	0	253	4	7	800	0	584	18	19
	500	212	512	17	15	1,000	0	786	19	40
	1,000	1,034	1,705	48	50	2,000	0	2,119	34	25
	50	0	30	0	2					
	100	0	59	3	2					
	200	0	118	3	5					
	300	0	206	5	10					
	400	0	292	6	8					
	500	0	555	15	16					
	1,000	0	2,823	44	60					
3 clienti locali	50	0	21	2	1	150	0	251	3	3
	100	0	50	3	1	300	0	310	6	5
	200	0	115	3	2	600	0	625	11	13
	300	0	203	5	9	900	0	792	13	18
	400	61	333	10	12	1,200	0	951	23	38
	500	279	673	13	22	1,500	0	1,250	40	35
	1,000	1,056	2,510	21	39	3,000	0	3,864	54	75

BUPT

	50	0	26	3	1					
	100	0	50	3	3					
	200	0	133	8	6					
	300	0	208	7	9					
	400	256	535	16	10					
	500	249	620	19	17					
	1,000	1,866	3,510	39	41					
	50	0	21	0	1					
	100	0	47	1	2					
	200	0	127	8	3					
	300	0	212	6	8					
	400	191	463	13	3					
	500	0	232	4	9					
	1,000	1,778	3,414	37	42					
4 clienti	50	0	21	1	1	200	0	346	2	7
locali	100	0	55	0	3	400	0	450	7	10
	200	0	125	4	4	800	0	732	11	13
	300	0	251	7	5	1,200	0	1,171	24	30
	400	78	343	10	5	1,600	0	1,506	24	32
	500	93	439	13	8	2,000	0	2,254	25	41
	1,000	509	2,181	34	28	4,000	0	5,164	90	98
	50	0	29	0	2					
	100	0	55	2	3					
	200	0	126	5	4					
	300	0	254	6	7					
	400	402	666	10	22					
	500	646	1,188	21	24					
	1,000	2,215	4,642	50	60					
	50	0	22	0	1					
	100	0	59	3	4					
	200	0	121	3	4					
	300	0	221	5	13					
	400	538	823	19	16					
	500	959	1,574	20	27					
	1,000	2,120	4,631	52	53					
	50	0	28	0	2					
	100	0	54	1	1					
	200	0	132	1	3					
	300	0	232	4	9					
	400	220	581	19	24					
	500	767	1,440	14	23					
	1,000	1,381	4,060	25	34					

1. Varianta distribuită cu transmisie in inel

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Un client la	50	0	147	0	1	50	0	145	1	0
distanţă	100	0	290	4	2	100	0	288	1	0
	200	0	580	11	9	200	0	578	1	2
	300	0	862	3	10	300	0	859	3	2
	400	0	1,144	5	4	400	0	1,142	4	3
	500	0	1,440	7	1	500	0	1,440	7	1
	1,000	0	2,868	23	10	1,000	0	2,865	5	11
2 clienti la	50	0	212	2	0	100	0	347	5	2
distanţă	100	0	442	0	4	200	0	534	8	6
	200	0	886	9	4	400	0	970	10	14
	300	0	1,381	13	13	600	0	1,592	12	15
	400	0	1,854	7	7	800	0	2,017	13	30
	500	0	2,284	9	11	1,000	0	2,451	22	25
	1,000	0	4,290	15	14	2,000	0	4,489	53	66
	50	0	206	1	1					
	100	0	435	0	4					
	200	0	880	0	5					

BUPT

	300	0	1,375	8	15					
	400	0	1,854	9	9					
	500	0	2,280	7	6					
	1,000	0	4,270	6	8					
3 clienti la distanta	50	0	292	2	0	150	0	456	3	4
	100	0	573	3	6	300	0	928	6	8
	200	0	1,176	7	4	600	0	1,485	5	9
	300	0	1,775	7	4	900	0	2,121	13	15
	400	0	2,355	11	13	1,200	0	2,675	14	16
	500	0	2,977	12	13	1,500	0	3,433	23	21
	1,000	0	5,951	18	13	3,000	0	7,702	44	58
	50	0	293	1	0					
	100	0	582	1	3					
	200	0	1,182	2	3					
	300	0	1,784	8	10					
	400	0	2,357	9	10					
	500	0	2,926	10	11					
	1,000	0	6,060	34	20					
	50	0	288	1	1					
	100	0	578	4	1					
	200	0	1,165	2	3					
	300	0	1,763	4	4					
	400	0	2,347	2	3					
	500	0	2,981	3	5					
	1,000	0	6,058	15	23					
4 clienti la distanță	50	0	275	1	3	200	0	767	5	7
	100	0	570	3	1	400	0	1,119	5	14
	200	0	1,161	2	3	800	0	1,684	13	19
	300	0	1,746	3	8	1,200	0	2,493	27	41
	400	0	2,245	6	4	1,600	0	3,146	31	47
	500	0	3,019	8	4	2,000	0	3,628	46	41
	1,000	0	6,263	15	10	4,000	0	7,131	60	73
	50	0	286	1	1					
	100	0	582	0	1					
	200	0	1,191	12	2					
	300	0	1,774	13	9					
	400	0	2,453	13	10					
	500	0	3,043	12	13					
	1,000	0	6,289	30	28					
	50	0	296	1	2					
	100	0	577	6	6					
	200	0	1,193	5	2					
	300	0	1,769	4	8					
	400	0	2,455	9	5					
	500	0	3,048	7	4					
	1,000	0	6,263	14	19					
	50	0	287	3	5					
	100	0	567	5	9					
	200	0	1,175	2	6					
	300	0	1,758	8	2					
	400	0	2,422	6	9					
	500	0	3,020	4	7					
	1,000	0	6,189	17	16					
Un client local	50	0	25	0	2	50	0	20	0	1
	100	0	35	2	3	100	0	33	1	3
	200	0	85	4	5	200	0	84	4	6
	300	0	161	12	10	300	0	161	3	9
	400	0	196	13	11	400	0	197	5	15
	500	0	264	14	11	500	0	265	8	13
	1,000	0	554	28	28	1,000	0	555	16	18
2 clienti locali	50	0	28	1	3	100	0	39	2	5
	100	0	52	4	0	200	0	83	1	3
	200	0	89	5	3	400	0	235	4	9

BUPT

	300	0	132	6	6	600	0	307	16	21
	400	0	183	5	13	800	0	416	13	28
	500	0	366	13	19	1,000	0	576	24	32
	1,000	0	975	17	25	2,000	0	1,248	33	39
	50	0	28	0	3					
	100	0	62	3	4					
	200	0	98	7	6					
	300	0	137	8	8					
	400	0	210	10	18					
	500	0	403	8	13					
	1,000	0	979	23	18					
3 clienti locali	50	0	16	1	1	150	0	248	3	5
	100	0	50	1	0	300	0	292	12	16
	200	0	106	3	3	600	0	530	19	19
	300	0	185	9	4	900	0	718	30	14
	400	0	256	9	10	1,200	0	863	19	37
	500	0	387	15	9	1,500	0	927	28	33
	1,000	71	997	19	27	3,000	0	1,783	55	64
	50	0	16	1	1					
	100	0	36	1	2					
	200	0	95	1	3					
	300	0	199	6	6					
	400	0	265	7	12					
	500	0	457	8	14					
	1,000	71	1,221	20	74					
	50	0	32	3	1					
	100	0	41	2	1					
	200	0	104	3	6					
	300	0	168	6	8					
	400	0	221	10	7					
	500	0	328	6	12					
	1,000	73	1,002	27	21					
4 clienti locali	50	0	22	3	1	200	0	339	5	5
	100	0	38	1	1	400	0	444	8	5
	200	0	103	5	4	800	0	623	25	24
	300	0	195	7	10	1,200	0	781	29	21
	400	0	281	9	11	1,600	0	1,089	31	33
	500	0	313	12	16	2,000	0	1,369	32	44
	1,000	234	1,357	18	32	4,000	0	3,069	67	84
	50	0	22	2	1					
	100	0	64	1	3					
	200	0	118	4	5					
	300	0	153	6	7					
	400	0	297	5	7					
	500	0	335	9	12					
	1,000	677	2,293	36	54					
	50	0	24	1	2					
	100	0	86	4	1					
	200	0	114	4	4					
	300	0	286	4	11					
	400	0	244	6	7					
	500	0	398	8	11					
	1,000	678	2,280	41	37					
	50	0	25	1	1					
	100	0	79	1	1					
	200	0	97	5	3					
	300	0	286	7	4					
	400	0	241	5	8					
	500	0	412	8	21					
	1,000	485	1,829	42	43					

BUPT

1. Varianta semidistribuită

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Un client la distanţă	50	0	82	0	0	50	0	82	0	0
	100	0	190	2	1	100	0	190	0	2
	200	0	454	1	2	200	0	454	1	2
	300	0	729	1	2	300	0	728	1	2
	400	0	956	1	2	400	0	954	0	2
	500	0	1,129	5	3	500	0	1,440	2	5
	1,000	0	2,321	9	14	1,000	0	2,865	4	11
2 clienti la distanţă	50	0	71	4	1	100	0	220	5	2
	100	0	235	0	2	200	0	338	8	16
	200	0	554	0	3	400	0	970	10	14
	300	0	918	6	8	600	0	978	12	15
	400	0	1,353	3	2	800	0	1,372	19	20
	500	0	1,606	4	5	1,000	0	1,702	26	20
	1,000	0	3,331	5	5	2,000	0	3,354	65	55
	50	0	73	0	1					
	100	0	276	0	1					
	200	0	484	2	2					
	300	0	904	0	2					
	400	0	1,319	9	17					
	500	0	1,528	2	7					
	1,000	0	3,232	1	5					
3 clienti la distanta	50	0	123	0	0	150	0	230	6	2
	100	0	316	0	1	300	0	743	6	8
	200	0	767	1	1	600	0	859	13	8
	300	0	1,251	1	0	900	0	1,289	14	20
	400	0	1,594	0	2	1,200	0	1,616	20	33
	500	0	1,936	4	2	1,500	0	2,309	23	36
	1,000	0	4,303	9	13	3,000	0	4,477	44	58
	50	0	124	0	1					
	100	0	316	0	3					
	200	0	721	1	4					
	300	0	1,261	5	2					
	400	0	1,619	11	12					
	500	0	2,057	8	15					
	1,000	0	4,262	9	10					
	50	0	92	0	0					
	100	0	301	0	1					
	200	0	730	1	1					
	300	0	1,141	2	3					
	400	0	1,455	1	3					
	500	0	1,939	2	3					
	1,000	0	4,197	4	7					
Un client local	50	0	100	0	0	50	0	98	0	0
	100	0	201	0	0	100	0	198	0	0
	200	0	400	0	0	200	0	397	0	0
	300	0	600	0	0	300	0	598	0	0
	400	0	802	0	0	400	0	799	0	0
	500	0	1,000	0	0	500	0	998	0	0
	1,000	0	2,002	0	0	1,000	0	2,000	0	0
2 clienti locali	50	0	139	0	0	100	0	197	0	0
	100	0	356	0	0	200	0	396	0	0
	200	0	765	0	0	400	0	798	0	0
	300	0	1,134	0	0	600	0	1,196	0	0
	400	0	1,572	0	0	800	0	1,594	0	0
	500	0	1,935	0	0	1,000	0	1,996	0	0
	1,000	0	3,947	0	0	2,000	0	3,998	0	0
	50	0	140	0	0					
	100	0	358	0	0					
	200	0	764	0	0					
	300	0	1,136	0	0					

BUPT

	400	0	1,576	0	0					
	500	0	1,937	0	0					
	1,000	0	3,947	0	0					
3 clienti locali	50	0	167	0	0	150	0	265	0	0
	100	0	369	0	0	300	0	457	0	0
	200	0	776	0	0	600	0	855	0	0
	300	0	1,151	0	0	900	0	1,259	0	0
	400	0	1,527	0	0	1,200	0	1,693	0	0
	500	0	1,935	0	0	1,500	0	2,075	0	0
	1,000	0	3,947	0	0	3,000	0	4,078	0	1
	50	0	192	0	0					
	100	0	394	0	0					
	200	0	796	0	0					
	300	0	1,202	0	0					
	400	0	1,598	0	0					
	500	0	1,994	0	0					
	1,000	0	4,004	0	0					
	50	0	165	0	0					
	100	0	375	0	0					
	200	0	778	0	0					
	300	0	1,185	0	0					
	400	0	1,576	0	0					
	500	0	1,981	0	0					
	1,000	0	3,981	0	0					
4 clienti locali	50	0	169	0	0	200	0	397	0	0
	100	0	357	0	0	400	0	554	0	0
	200	0	747	0	0	800	0	925	0	0
	300	0	1,156	0	0	1,200	0	1,372	0	0
	400	0	1,553	0	0	1,600	0	1,744	0	0
	500	0	1,910	0	0	2,000	0	2,166	0	1
	1,000	0	3,951	0	0	4,000	0	4,151	1	1
	50	0	198	0	0					
	100	0	394	0	0					
	200	0	803	0	0					
	300	0	1,194	0	0					
	400	0	1,594	0	0					
	500	0	1,969	0	0					
	1,000	0	4,001	0	1					
	50	0	195	0	0					
	100	0	362	0	0					
	200	0	801	0	0					
	300	0	1,197	0	0					
	400	0	1,601	0	0					
	500	0	2,000	0	0					
	1,000	0	4,004	0	0					
	50	0	181	0	0					
	100	0	397	0	0					
	200	0	785	0	0					
	300	0	1,153	0	0					
	400	0	1,559	0	0					
	500	0	1,995	0	1					
	1,000	0	3,947	0	1					

Se observă că:

(1) pentru varianta distribuită broadcast se obţin cei mai buni la transmisia mesajelor, atîta timp cît serverul nu este sufocat şi nu se fac retransmisii (retransmisiile se fac în situaţia în care în care serverul returnează o eroare de tip EBUSY);

(2) deoarece s-a ales pentru dimensiunea bufferului de mesaje din token o dimensiune de cel puţin de două ori mai mare decît numărul maxim de staţii care se pot conecta la sistem, pentru varianta distribuită cu transmisie în inel nu se poate obţine situaţia în care nu mai este loc pentru o cerere în token.

(3) timpul obţinut pentru varianta în inel este cel mai slab; rezultatul este însă compensat de fapul că în această variantă nu se poate obţine situaţia de "server de comunicaţie sufocat";

(4) pentru varianta semidistribuită se obţin timpi intermediari între cele două variante distribuite.

BUPT

(5) în cazul cînd în sistem există conectată o singură stație, cea mai rapidă variantă este varianta distribuită cu transmisie în inel, deoarece serverul de comunicație nu se mai execută (returnează o confirmare pentru orice cerere de la managerul de grupuri). În acest caz, se observă că cea mai proastă comportare o are varianta semidistribuită, pentru care trebuie să lucreze două servere.
(6) în cazul cînd în sistem există conectată o singură stație, în varianta cu transmisie in inel, începînd de la 4 procese locale au loc retransmisii locale către managerul de grupuri, datorită depășirii capacității bufferului de mesaje.
În figurile următoare se prezintă centralizat aceste concluzii.

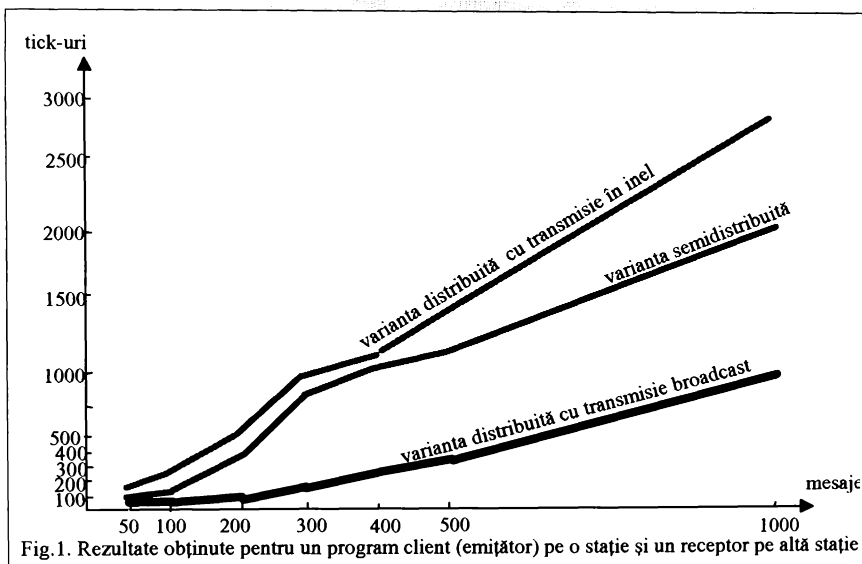


Fig.1. Rezultate obținute pentru un program client (emițător) pe o stație și un receptor pe altă stație

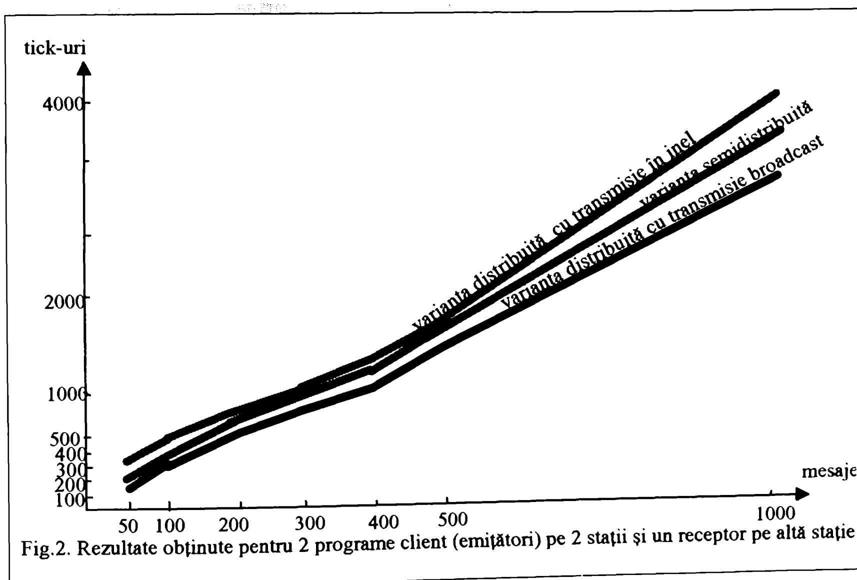


Fig.2. Rezultate obținute pentru 2 programe client (emițători) pe 2 stații și un receptor pe altă stație

BUPT

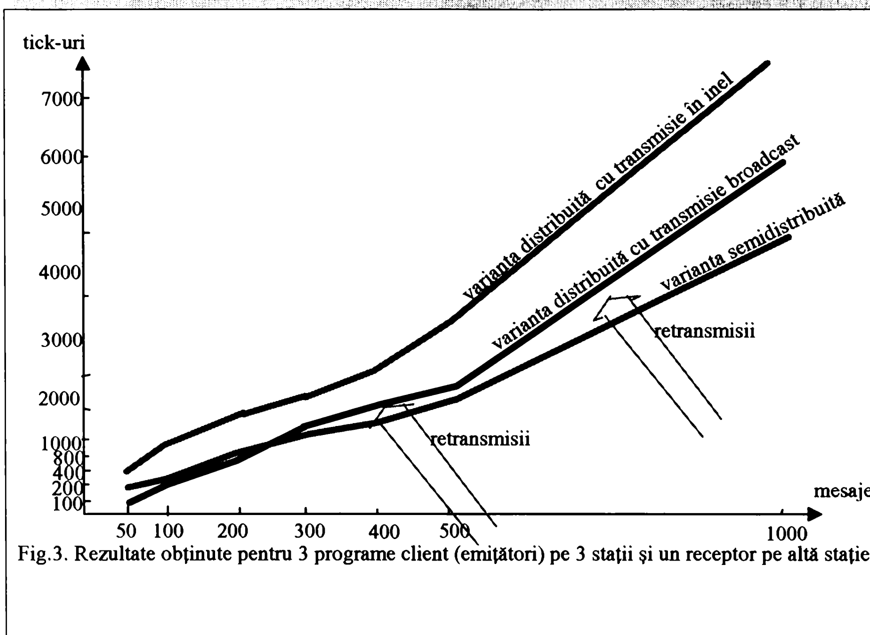


Fig.3. Rezultate obţinute pentru 3 programe client (emiţători) pe 3 staţii şi un receptor pe altă staţie

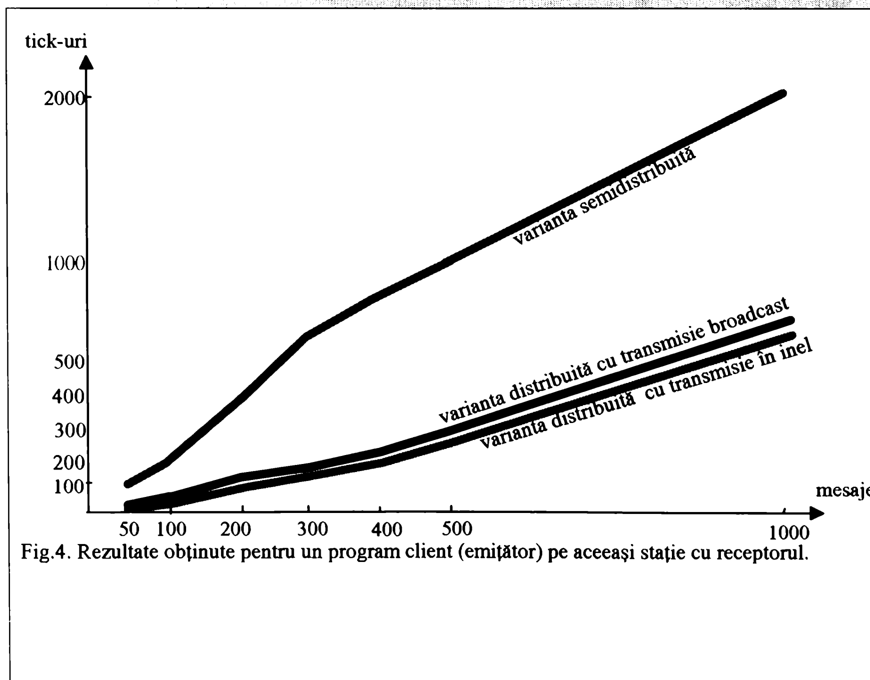


Fig.4. Rezultate obţinute pentru un program client (emiţător) pe aceeaşi staţie cu receptorul.

BUPT

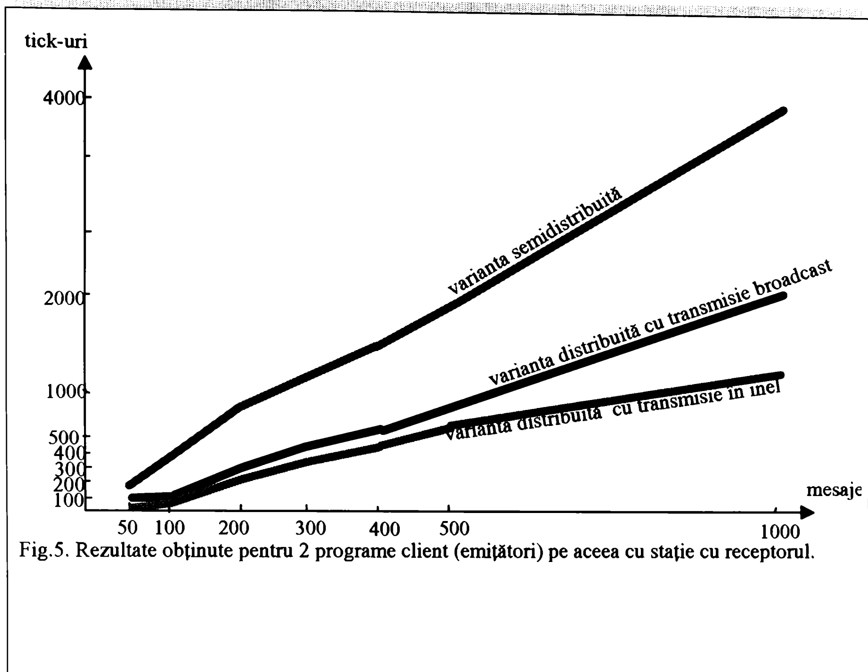


Fig.5. Rezultate obţinute pentru 2 programe client (emiţători) pe aceea cu staţie cu receptorul.

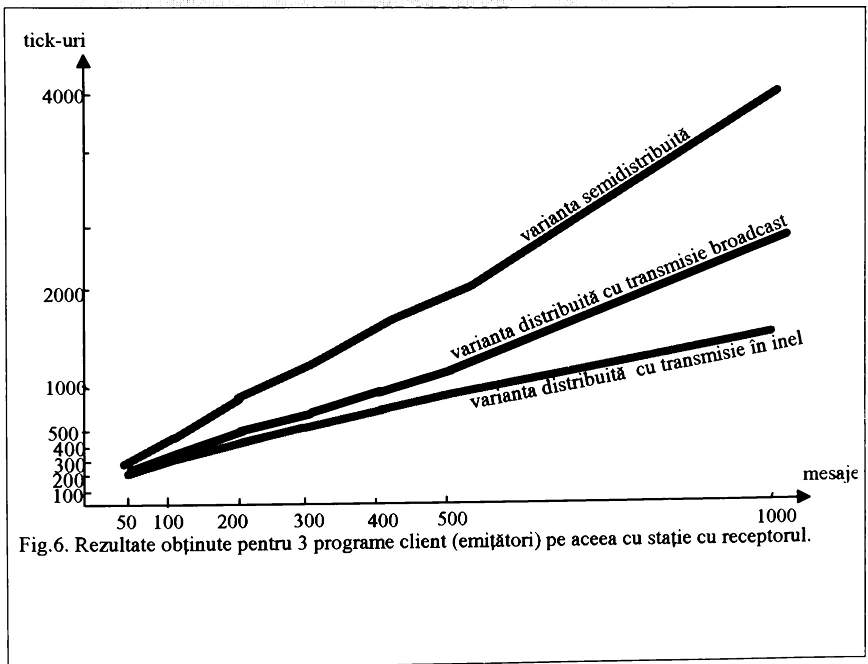


Fig.6. Rezultate obţinute pentru 3 programe client (emiţători) pe aceea cu staţie cu receptorul.

BUPT

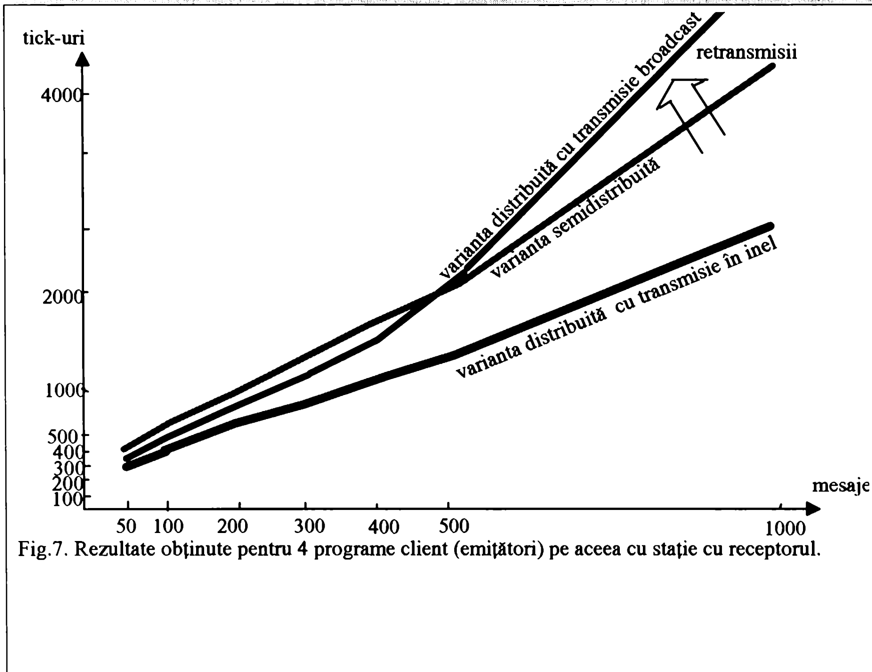


Fig.7. Rezultate obţinute pentru 4 programe client (emiţători) pe aceea cu staţie cu receptorul.

BUPT

Bibliografie.

[1] [ABB93] Assenmacher H., Breitbach T., Buhler P., Hubsch V., Schwarz R.: "PANDA-Supporting Distributed Programming in C++", in *Object Oriented Programming*, pp361-383, Springer Verlag, 1993

[2] [ABN85] Almes G.T., Black. A.P., Lazawska E.D. and Noe J.D.: "The Eden System: A Tehnical Review", *IEEE Transaction on Software Engineering* SE-11 (1), pp. 43-59,1985

[3] [AC93] Atkins N.S., Coady M.S.: "Adaptable Concurrency Control for Atomic Data Types", *ACM Transaction on Computer Systems*, Vol.10, No.3, pp 190-225, 1993

[4] [ACF87] Artsy Y. Chang H.Y. and Finkel R.: "Interprocess Communication in Charlotte", *IEEE Software*, pp 22-28, 1987

[5] [ADB87] Ahamad M., Dasgupta P., Le Blanc R.J., and Wilkis, C.T. : "Fault taulerant computing in object-based distributed systems", *IEEE 6th Simposium of Reliablity in Distributed Software and Database Systems*, 1987

[6] [ALP90] Ancilotti, P., Lazzerini, B., Prete, C., A, and Sacchi, M. :"A Distributed Commit Protocol for a Multicomputer System", *IEEE Transactions on Software Engineering*, vol.16, No.5, pp 718-725, 1990

[7] [And90] Anderson A.: "Multiple Processing. A System Overview." , McGrawHill 90

[8] [And91] Andrews G.R.: "Paradigms for Process Interaction in Distributed Programs", *ACM Computing Surveys*, vol 23, 1, 49-90, 1991

[9] [AS83] Andrews G.R., and Schneider F.B.: " Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, 15(1), pp 3-49, 1983

[10] [AW94] Attiya, H., Walch, J.: "Sequential Consistency versus Linearizability", *ACM Transactions on Computer Systems*, Vol.12, No.2, pp 91-122, 1994

[11] [Bac89] Bach M.: "The Design of the Unix Operating System", Prentice Hall, 1989

[12] [BBK91] Balter R., Banatre J.B., Krakowiasks : "Construction des systemes repartis", Ed. Collection Didactic, Inria 91

[13] [C87] Bernshad B.N., Ching D.T., Lazawska E. D. , Sanislo E.D. and Schwarz M.; "A Remote Procedure Call Protocol Facility for Interconnecting Heterogeneous Computer Systems", *IEEE Transactions on Software Engineering* SE-13 (8), pp 880-894, 1987

[14] [BDW89] Barbacci M., Doubleday D., Weinstock Ch.: "The Durra Language and Runtime Environment: Tools for PMS-level Programming", *Working Conference on Descentralized Systems*,Proceedings, pp1-16, Lyon, 1989

[15] [Bel 89] Belzile, C., Coulas M., MacEwen G.H., Marquis G.: "RNET: A Hard Real Time Distributed Programming System", *Proceeding of the 1989 Real-Time Systems Symposium*, pag 2-13., 1989

[16] [BHG87] Bernstein Ph., Hadzilacos V., Goodman N.,: "Concurrecy Control and Recovery in Database Systems", Addison Wesley 87

[17] [BF80] Bullis K., Franta W.: " Implementation of Evencounts in a Broadcast Network", Computer Networks, 4, 57-69, 1980

[18] [BF85] LeBlanc T., Friedberg S.: "HPC: A Model of Structure and Change in Distributed Systems", *IEEE Transaction on Computers*, vol. C-34, pp 1114-1130, dec. 1985

[19] [BG81] Bernstein P.A. and Goodman N. : "Concurency control in distributed database systems', *ACM Computing Surrveys.*, 13, 2 (Jun), 185-221, 1981

[20] [BGH93] Bever M., Geihs K., Heuser L., Muhlhauser M., Schill A.: " Distributed Systems, OSF DCE and Beyond", in *Distributed Systems*, pp1-20, Springer Verlag, 93

[21] [Bir91] Birmann K.: "Fault-Tolerance in Sixth Generation Operation Systems", in *Operating Systems of the 1990s and Beyond*, pp154-161, Springer-Verlag, 1991

[22] [BJ87] Birman K.P., Joseph T.A.: "Reliable Communications in The Presence of Failures", *ACM Transaction on Computer Systems*, vol.5, Nr.1, 46-47, Feb.87

[23] [BJ90] Birman, K.P., Joseph, T.A: "Communication support for reliable distributed computing", *Fault Taulerant Distributed Computing*, Springer-Verlag, 1990

[24] [BK85] Bancilhon F.K., and Korth H.:"A Model of CAD Transactions", *Proceeding of the 11th International Conference on Very Large Databases*, pp.25-33, 1985

[25] [BKT85] Brown M.R., Kolling K.N., and Taft, E.A. :"The Alpine file system", *ACMTOCS*, vol.3, no.4, pp.261-193, Nov. 1985

BUPT

BUPT

[26] [BK91] Barghouti N.S., and Kaiser G.E. :"Concurrency Control In Advanced Database Applications", *ACM Computing Surveys*, vol.23, No.4, pp 269-294, 1991

[27] [BN84] Birell A. D. and Nelson B. J.: "Implementing Remote Procedure Calls", *ACM Transaction on Computers Systems* 2 (1) pp 39-59, Feb 1984

[28] [BT84] Brancha G., Toneg S.: "A distributed algorithm for generalized deadlock detection", *Proceedings of the Symposium on Principles of Distributed Computing*, ACM, New York, pp 285-301, 1984

[29] [CC91] Chin R., Chonson S.: "Distributed Object-Based Programming Systems", *ACM Computing Surveys*, Vol.23, No.1, pp 91-124, March 1991

[30] [CDI90] Ciciani B., Dias D. Iyer, B., Yu, P. :"A Hibrid Distributed Centralized System Structure for Transaction Processing", *IEEE Transactions on Software Engineering*, vol.16, No.8, pp 791-805, 1990

[31] [CG87] Cellary, Gelenbe.: Concurrency Control in Distributed Database Systems", Addison Wesley 87

[32] [Cha82] Chandy K.M.: "A Mutual Exclusion Algorithm for Distributed Systems", *Techn. Rep.* University of Texas

[33] [Che84] Cheriton D.R.: " The V Kernel : A Software Base for Distributed System", *IEEE Software*, pp 19-42,1984

[34] [CL85] Chandy K.M., and Lamport, L.: "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Programming Language Systems*, 3, 1, 63-75, Feb. 1985

[35] [CL87] Chin L., Liu M.T.: "An optimistic concurrency control mechanism without freezing for distributed database systems", in *Proceeding Conf. on Data Engineering*, pp.322-329, LA, 1987

[36] [CO82] Ceri S., Owicki S.: "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", *Proc. 6th Berkeley WorkShop on Distributed Data Management and Computer Networks*, pp.117-130, Berkeley, 1982

[37] [CS91] Comer D., Stevens D., "Internetworking with TCP/IP", Prentice Hall, 1991

[38] [CM82] Chandy K.M., and Misra, J. : "A Distributed algorithm for detecting resource deadlocks in distributed systems", *Proceedings of the Symposium on Principles of Distributed Computing*, ACM, New York, pp 157-164, 1982

[39] [CM84] Chang J., Maxemohuk N.F.: "Reliable Broadcast Protocols", *ACM Transaction on Computer Systems*, vol.2, Nr.3, 251-273, Aug.84

[40] [CM86] Chandy K.M., and Misra J.: "An exemple of stepwise refinement of distributed programs: Quiescence detection", *ACM Transactions on Programming Language Systems*, 8, 3, 326-343, July 1986

[41] [CMH83] Chandy K.M., Misra J., and Haas L.M., : "Distributed deadlock detection", *ACM Transactions on Computer Systems*, 1, 2, pp 147-156, May 1983

[42] [CPS93] Cabrera L.F., McPherson J. Schwarz, P.M., Wyllie, J.: "Implementing Atomicity in Two Systems: Techniques, Tradeoffs and Experience", *IEEE Trans. on Software Engineering*, Vol.19, No.10, pp 950-601, Oct 1993

[43] [CP84] Ceri S., Pellagatti, G.: "Distributed Databases: Principles and Systems", McGraw-Hill, 1984

[44] [CZ83] Cheriton D.R. and Zwaenepoel W.: "The Distributed V Kernel and its Performance for Disklles Wokstations", *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, pp. 129-140, 1983

[45] [Cri89] Crichlow A.: " Introduction to Distributed and Parallel Computing", McGraw-Hill, 1989

[46] [CTM92] Cristea V., Tapus N., Moisa T., Damian V.: " Retele de calculatoare", Ed Teora 1992

[47] [Cri96] Cristian F.: "Synchronous and Asynchronous Group Communication", *Communication on the ACM*, vol 39, no 4, april 1996

[48] [Dav81] Davies D. W.: Distributed Systems-Architectures and Implementation", Lectures Notes in Computer Science, Springer Verlag, Berlin 1981

[49] [Dav94] Davis R.: " Windows NT Network Programming", Addison-Wesley 1994

[50] [Dei90] Deitel H.: "An Introduction to Operating System", Addison Wesley 1990

[51] [DG89] Delattre E., Geib J.M.: "OMPHALE, an active objects-based distributed operating system", *Working Conference on Decentralized Systems*, Proceedings, Lyon 89, pag 327-340

[52] [Dil93] Dilley J.: "Object-Oriented Distributed Computing With C++ and OSF DCE", in *DCE-The OSF Distributed Computing Environment Client Server Mode and Beyond*, pp256-266, Springer-Verlag, 1993

[53] [DM96] Dolev D., Lalki D.: "The Transis Approach to High Availability Cluster Communication", *Communication on the ACM*, vol 39, no 4, pp54-63, april 96

BUPT

[54] [DVP88] Dumitrescu V., Teodor V., Paiu O., Stefanescu V.: " Initiere in informatica distribuita", Ed. tehnica, Bucuresti, 1988

[55] [ED88] Elmagarmid A.,K., and Datta A.,K.: "Two-Phase Deadlock Detection Algorithm", *IEEE Transactions on Computers*, Vol.37, No.11, pp 1454-1458, 1988

[56] [EGT76] Eswaran K.P., Gray J.N., Lorie R.A., Traiger, I.L.: "The Notion of Consistency and Predicate locks in a Database System", *Communication on the ACM*, 19(11), 624-633, 1976

[57] [FHR93] Fazzolare M., Humm B., Ranson D.: "Object Oriented Extendibility in Hermes/ST, a Transactional Distributed Programming Environment", in *Object Based Distributed Programming*, pp240-262, Springer Verlag, 1993

[58] [For88] Fortier P.: " Design of Distributed Operating Systems", McGraw Hill, 1988

[59] [FSK89] Finkel R., Scott M.L., Kalsow W.K., Artsy Y. and Chang H.Y.: " Experience with Charlotte : Simplicity and Function in a Distributed Operating System", *IEEE Transactions on Software Engineering*,SE - 15, 6, pp 676-685, 1989

[60] [GDT92] Grassi V.,Donatiello L.,Tucce S.: "On the Optimal Checkpointing of Critical Tasks and Transactions", *IEEE Transactions on Software Engineering,* vol 18, no 1, 1992

[61] [Gue94] Guerraoi R.: "Atomic Object Composition", in *Object Oriented Programming*, pp118-139, Springer Verlag, 1994

[62] [GM83] Garcia-Molina H. :"Using semantic Knowledge for transaction processing in a distributed database", *ACM Transactions on Database Systems*, 2 (June), pp.186-213, 1983

[63] [GMo82] Garcia Molina H.: "Election in a Distributed Computing System", *IEEE Transactions on Computers,* C-31(1), 48-59, 1982

[64] [GS91] Garcia Malina H., Spanster A.: "Ordered and Reliable Multicast Communication", *ACM Transaction on Computer Systems*, vol. 9 nr.3, 242-271, Aug.1991

[65] [Gif79] Gifford D.K., "Weighteid Voting for Replicated Data"; *Proceedings of the Seventh Symposium on Operating Systems Principles,* ACM, 150-162, 1979

[66] [Gie91] Gien M.: "Next Generation Operating Systems Architecture", in *Operating Systems of the 1990s and Beyond*, pp227-232, Springer-Verlag, 1991

[67] [GS80] Gligor V., Shattick S.: "On deadlock detection in distributed databases", *IEEE Transactions on Software Engineering*, SE-6, 5 Sept.1980

[68] [Gos91] Goscinski A.: "Distributed Operating Systems- The logical design", Addison Wesley, 1991

[69] [Hag87] Hagmann R. :"Reimplementing the Cedar file system using logging and group commit", *Proceedings of the 11th SOSP*, ACM, pp.155-162, Nov. 1987

[70] [Hal88] Halshall F.: "Data Communications Computer Network and OSI", second edition, Addison Wesley, 1988

[71] [Hin96] Hinden R.M. : " IP Next Generation Overview", *Communication on the ACM*, vol. 39, No.6, Jul.96

[72] [HJP87] Helary J., Jard C., Planzean, N., and Rayna, M., 1987 :"Detection of stable properties in distributed application", *Proceedings of the Symposium on Principles of Distributed Computing*, ACM, New York, pp 125-136, 1987

[73] [HT83] Haerder T., Reuter A.:"Priciples of Transactions-Oriented Database Recovery", *ACM Computing Surveys*, vol.15, No.4, pp 287-289, Dec. 1983

[74] [HSE91] Hoffman F., Schlenk P., Eirich T., "Encapsulation and Interaction in Future Operating Systems", in *Operating Systems of the 1990s and Beyond*, pp241-245, Springer-Verlag, 1991

[75] [HW88] Herlity M.P., and Weihl W.E.: "Hybrid Concurrency for Abstract Data Types", *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, ACM Press, pp.201-210, 1988

[76] [JB89] Joseph T.A. and Birman K.P.: "Reliable Broadcast Protocols" in *Distributed Systems*, ed. Mullender S.J., ACM Press, pp293-316, 1989

[77] [JLH88] Jul E., Levy H., Hutchinson N. and Black A.: "Fine-grained Mobility in the Emerald System", *ACM Transactions on Computer Systems* 6(1), pp 109-133, 1988

[78] [Jon79] Jones A.: "The Object Model: A Conceptual Tool for Structuring Software", in *Operating Systems-An Advanced Course,* Springer-Verlag,1979

[79] [JRT85] Jones M.B., Rashid R.F. and Thompson M.R.: "Matchmaker: An Interface Specification Language for Distributed Processing", *Proceedings of the 12th ACM Symposium on Principles on Programming Language*, pp. 225-235, 1985

[80] [JW93] Johnson B. David, Zwaenepoel W.: "The Peregrine High-Performance RPC Systems", *Software Practice and Experience*, vol. 23(2)[119]

[81] [Jur84] Jurca I.: "Sisteme de operare", curs, Reprografie, Universitatea Politehnica Timisoara, 1984

[82] [Jur92] Jurca I: "Programarea orientata pe obiecte in limbajul C++", ed. Eurobit, Timisoara,1992

BUPT

[83] [Kap87] Kapp E. : "Deadlock Detection in Distributed Databases", *ACM Computing Surveys*, Vol.19, Nr.4, Dec.1987

[84] [Koh81] Kohler W.H.: "A Survey of Techniques for Syncronization and Recovery in Distributed Computer Systems", *ACM Computing Surveys*, 13, 2, 149-183, 1981

[85] [KR81] Kung H.T., and Robison J.T.: "Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 6(2), pp.213-226, 1981

[86] [KS89] Kuman A., Stonebraker M.: "Performance Consideration of an Operating System Transaction Manager", *IEEE Transactions on Software Engineering*, vol.15, No.6, June 1989, pp 705-715

[87] [Lag79] Lagally, K: "Synchronization in a Layered System", in *Operating Systems-An Advanced Course*, Ed. Flynn, M. J., Gray, J. N., Jones, A. et al. , Springer-Verlag 1979

[88] [Lam78] Lamport L.: Time, Clocks and the Ordering of Events in a Distributed System", *Communication on the ACM*, 21, 7 July 78

[89] [Lam81] Lampson E.W. : "Atomic Transactions", *Distributed Systems Architecture and Implementation*, Springer-Verlag, Berlin 246-264, 1981

[90] [LG85] Lee I., Goldwasser, S.M.: "A Distribute Testbed for Active Sensory Processing", *International Conference on Robotics and Automation*, pp925-930, 1985.

[91] [LLA81] Lazawska E.D., Levy H.M., Almes G.T., Fisher M.I., Fowler R.I., Vestal S. C.: "The Architecture of the Eden Systems", *Proceedings of the Eight Symposium on Operating Systems Principles*, Pacific Grove, California, pp. 148-159, 1981

[92] [LRM93] Lashkari Y., Ramachandar V., Malpani S., Mehndiratta S.L.: "Vartalaap: a Distributed Multicast Communication System", *Software Practice and Experience*, vol.23(7), 799-811, Jul. 1993

[93] [Lan83] LeLann G.: "Synchronization", in *Local Area Networks: An Advanced Course*, Lectures Notes in Computer Science, Springer-Verlag, pp361-395, 1983

[94] [LCJ87] Liskov, B., Curtis, D., Johnson, P. and Scheifler, R.: "Implementations of Argus", *ACM Proceedings 12th Symposium on Operating System Principles*, pp 111-122, 1987

[95] [Lis88] Liskov B.: "Distributed Programming in Argus", *Communication on the ACM* 31. 3, 300-312, March1988

[96] [Lis90] Liskov B.: "Argus", *Fault Taulerant Distributed Computing*, Ed. Simons, B., Spector, A., Springer-Verlag, pp108-115, 1990

[97] [LS83] Liskov B. and Scheifler R.: "Guardians and Actions: Linguistic Support for Robust. Distributed Programs", *ACM Transactions on Programming Languages and Systems* 5 (3), pp 381-404, July 1983

[98] [LS90] Levy E., Silberschatz A.: "Distributed File System: Concepts and Examples", *ACM Computing Surveys*, vol 22, No 4, pp 321-372, 1990

[99] [LM92] Lungu M., **Musatescu C**.: "Sistem in timp real pentru implementarea unei metode de calcul ierarhizat", *Simpozionul National de Teoria Sistemelor, Robotica, Calculatoare si Informatica de proces* (SINTES 6), ed. a 6-a Craiova, 28-29 martie 1992, pp82-88

[100] [Lyn83] Lynch N.A. :"Multilevel atomicity: A new corectness criterion for database concurrency control", *ACM Transactions on. Database Systems.*, 8,4, pp.484-502, 1983

[101] [Mar87] Martin B. :"Modelling concurrent activities with nested objects:, *Proceedings of the 7th International Conference on Distributed Systems*, IEEE Computer Society Press, pp.432-439, 1987

[102] [MK83] Magee J.,Kramer, J.: "Dynamic Configuration for Distribute Real-Time Systems", *Proceedings of the 1983 Real-Time Systems Symposium*, pag 277-288, 1983

[103] [Moc93] Mock M.: "DCE++. Distributed C++ Objects Using OSF DCE", in *Distributed Computing Environment"*, pp242-256, Springer Verlag, 1993

[104] [Mos85] Mos J.E.B. :"Nested Transaction: An approach to reliable distributed computing", The MIT Press, Cambridge, March, 1985

[105] [ML83] Mohan C., Lindsay B.: "Efficient commit protocols for the tree of Processes model of distributed transactions", *ACM Proceedings of the 2nd SIGALT/ SIGOPS Symposium of Principles of Distributed Computing*, pp 76-88, Aug.1983

[106] [MOO87] Maekawa M., Oldehoeft A.E., Oldehoeft R.R.: "Operating System Advanced Concepts", The Benjamin Cummings Publishing Co., 1987

[107] [MM79] Menasce D., and Muntz R.: "Locking and Deadlock Detection in Distributed Databases", *IEEE Transactions on Software Engineering* SE- 5, 6, 3 May 1979

[108] [MPP87] Miller B.P., Presotto D.L., Powel M.L., DEMOS/MP: "The Development of a Distributed Operating System", *Software-Practice and Experience*, 17 (4), pp. 277-290, 1987

BUPT

[109] [MMA96] Moser L.E., Melliar P.M., Agarwal A.D., Budhia R.K., Lingley-Papadopoulos C.A.: "Totem: A Fault-Tolerant Multicast Group Communication System", *Communication on the ACM*, vol 39, no 4, pp54-63, april 96

[110] [Mus92] **Musatescu C.**, Marian Gh., Padeanu L.: "Sistemul de operare SINIX-Modalitate de implementare a sistemului de operare UNIX.", *Simpozionul National de Teoria Sistemelor, Robotica, Calculatoare si Informatica de proces (SINTES 6)*, ed. a 6-a Craiova, 28-29 martie 1992, pp178-184

[111] [Mus93a] **Musatescu C.**, Mocanu M., Badica C.: "Symmetric and Unsymmetric Cryptosystems in Distributed UNIX-like Systems", *The International Conference on Applied and Theretical Electrotechnics*, (ICATE 93)18-20 nov. 93, Craiova, pp 315-320

[112] [Mus93b] **Musatescu C.**, Mocanu M., Badica C.: "Authentication and Encryption in UNIX-like Systems", *Romanian Open Systems Event ROSE-93* , Cluj 29-30 sept,1993

[113] [Mus93d] **Musatescu C.**: "Stadiul actual al dezvoltarii sistemelor distribuite", referat doctorat, Timisoara 1993

[114] [Mus94b] **Musatescu C.**: "Concepte si metode folosite in proiectarea aplicatiilor distribuite", *Simpozionul National de Teoria Sistemelor, Robotica, Calculatoare si Informatica de proces* (SINTES 7), ed. a 7-a Craiova, 20-23 mai 1994, pp 30-36

[115] [Mus94c] **Musatescu C.**: "Controlul concurentei in aplicatii complexe", *Simpozionul National de Teoria Sistemelor, Robotica, Calculatoare si Informatica de proces* (SINTES 7), ed. a 7-a Craiova, 20-23 mai 1994, pp 36-44

[116] [Mus94d] **Musatescu C.**: "Aplicatii in retele locale de calculatoare", Referat doctorat, Timisoara, 1994

[117] [MMG94] Marian Gh., **Musatescu C.**, Grosu M., Lungu M.: "Sistem distribuit pentru dirijarea optima a circulatiei in intersectii, sincronizarea semafoarelor de pe fluxurile de circulatie si supravegherea functionarii acestora", grant no 3006/14c/B9/1994, beneficiar Ministerul Invatamintului, 1994

[118] [Mus95a] **Musatescu C.**: "Support for Distributed Transaction in Distrib", *10-th International Conference on Control Systems and Computer Science*, Bucharest, 24-26 May 1995, pp287-291

[119] [Mus95b] **Musatescu C.**: "Distrib-A Distributed Object-Oriented System Based on Transaction Processing", *5-thSymposium Automatic Control and Computer Science*, Iasi 26-27 oct 1995, pp157-163

[120] [Mus95c] **Musatescu C.**: "Sistemul de operare Novell NetWare 3.11. Apeluri sistem", Reprografia Universitatii din Craiova, 1995

[121] [Mus95d] **Musatescu C.**: "Gestionarea fisierelor in retele de calculatoare", *Else*, no 5, pp2-19, dec 1994

[122] [Mus96a] **Musatescu C.**, Burdescu D.: "DOSTP-A Distributed Object Oriented System", The International Conference of tehnical Informatics, Timisoara Conti-94, 14-16 nov 1996, pp 25-33

[123] [Mus96b] **Musatescu C.**, Burdescu D.: "The Client Server Model in a Distributed Object Oriented System Based on Transaction Processing", *International Symposium On Systems Theory*, SINTES 8, Craiova 6-7 iunie 1996, pp 21-28

[124] [MMB96] Marian, **Musatescu C.**, Budica I: "A Strategy for Traffic Control System", *International Symposium On Systems Theory*, SINTES 8, Craiova 6-7 iunie 1996, pp143-152

[125] [Mus96d] **Musatescu C.**: "A Novell Implementation of a Distributed Object Oriented System Based on Transaction Processing", *International Symposium On Systems Theory*, SINTES 8, Craiova 6-7 iunie 1996, pp 180-188

[126] [Mus96e] **Musatescu C.**, Mocanu M., Olteanu C.: "TDL: A Run-Time Environment for Connecting Tasks in a Unix Local NetWork", *International Symposium On Systems Theory*, SINTES 8, Craiova 6-7 iunie 1996, pp309-316

[127] [Mus96f] **Musatescu C.**: "Aspecte ale proiectarii sistemelor de operare", *Tech. Rep. Tempus*,Reprografia Univ. Craiova, 1996

[128] [Mus96g] **Musatescu C.**, Burdescu D.: " The Design and the Implementation of a DOSTP- A Distributed Object Oriented System Based On Transactions Processing", *Facta Universitatis, series: Electronics and Energetics*, vol 9, no 2, 1996, pp 149-172, Nis, Yugoslavia

[129] [Mus96i] **Musatescu C.**: "Designing a Distributed Object Oriented System", *Research Rep.*, Technological Educational Institute of Piraeus, Faculty of engineering, Department of Computer Systems, Athens, Greece, may 1996

[130] [Mus96j] **Musatescu C.**: "Aspecte ale modelului client server in programarea distribuita", *Else*, no 9, pp33-52, 1996

[131] [Mus97] **Musatescu C.**, Burdescu D.,Hamburg I.: "Object Oriented System for Distributed Transactions during the Development of a Product within a group of Companies", *1-st*

BUPT

International Conference on Engineering Design and Automation, Bangkok, Thailand, March 18-21, 1997

[132] [Mull89] Mullender S.J., *Distributed Systems*, ACM Press, 1989

[133] [Mull89b] Mullender S.J., " Interprocess Communications", in *Distributed Systems*, ed. Mullender S.J., pp37-63, ACM Press, 1989

[134] [Mul90] Mullender, S. J.: "Amoeba-A Distributed Operating System for the 1990s", *Computer Magazine*, May 1990

[135] [Neh91] Nehmer J.: "The Immortality of Operating Systems or Is Research in Operating Systems Still Justified?", in *Operating Systems of the 1990s and Beyond*, pp77-82, Springer-Verlag, 1991

[136] [Nut92] Nutt G.: "Centralized and Distributed Operating Systems", Prentice Hall, 1992

[137] [OCD88] Ousterhout J.K., Cherensen A.R., Douglish F., Nelson M.N. and Welch B.B.: "The Sprite Network Operating system", *Computer*, 21(2), pp 23-26,1988

[138] [Pow83] Powell M.L. and Miller B.P.: "Process Migration in DEMOS/MP", in *Proceedings of the 9th ACM Symposium on Operating System Principles*, Bretton Woods, New Hampshire, pp 110-119,1983

[139] [PSU86] Pradel U., Schlageter G., and Unland R. :"Redesign of optimistic methods: Improving performance and availablity", *Proceedings of the 2nd International Conference on Data Engineering IEEE Computer Society Press*, pp.466-473, 1986

[140] [PG93] Paunescu F., Golesteanu D. P., "Sisteme cu prelucrare distribuita si aplicatiile lor", Ed tehnica 93

[141] [RA81] Ricard, G., Agrawala R. K.: "An optimal Algorithm for Mutual Exclusion in Computer Network", *Communication on the ACM*,22(2), 24(1), 9-17, 1981

[142] [Ras81] Rashid R.F.: "An Inter-Process Communication Facility for UNIX", *Local networks for Computer Communications*, Springer-Verlag, pp 319 -354, 1981

[143] [Ras88] Rashid R.F.: "Machine - Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Transactions on Computers*, C - 37(8), pp 896 - 908, 1988

[144] [Ray87] Raymond K.: "Multiple Entries with Ricard and Agrawala Distributed Mutual Exclusion Algorithm", Techn. Report, Department of Computer Science, The University of Queensland, 1987

[145] [RB88] Ramamritham K., Badrinat B.R. : "Syncronizing Transactions on Objects:, *IEEE Transactions on Computer*, Vol.37, No.5, 541-549, May 1988

[146] [RBM96] Robert van Renesse, Birman K.P., Maffeis S.: " Horus: A Flexible Group Communication System", *Communication on the ACM*, vol 39, no 4, pp54-63, april 96

[147] [RK79] Reed D.P., Kanodia R. K.: "Syncronization with Eventcounts and Sequencers", *Communication on the ACM*, 22(2), 115-123, 1979

[148] [RSB90] Ramanthan P., Shin K.G. and Butler R.W.: "Fault Taulerant Clock Synchronization in Distributed Systems", *IEEE Computer*, vol 23, pp 33-42, oct 1990

[149] [SBD86] Spector A.Z., Bloch, J.J., Daniels, D., Dravers D.S., Duchamp, D., Eppinger J.L., Menes S.G., and Thomson D.S. : "The Camelot Project", *Tech. Rep. CMU-CS-86*, 166, Departament of Computer Science, Carnegie Mellon University, Pittsburgh, Penn., 1986

[150] [SC91] Shrivastava S., McCue D.: "Operating System Support for Object Oriented Distributed Systems", in *Operating Systems of the 1990s and Beyond*, pp256-257, Springer-Verlag, 1991

[151] [SDE85] Spector A.Z., Daniels D.S, Duchamp D., Eppinger J.L., and Pausch R.: "Distributed Transactions for Reliable Systems", *Tech. Rep. CMU-CS-85*, 117, Departament of Computer Science, Carnegie Mellon University, Pitsburgh, Penn., 1985

[152] [SGA87] Salem K., Garcia-Molina H., and Alonso R.: "Altruistic locking: A strategy for coping with long-lived transactions", *Proceedings of the 2nd International WorkShop on High Performance Transaction Systems*, pp.19.1-19.24, 1987

[153] [SH82] Shoch J.F. and Hupp J.A. : "The Worm' Programs - Early Experience With a Distributed Computation", *Communications of the ACM*, 25,3, pp 173-80, 1982

[154] [SK89] Stonebraker M., Kumar A.: "Performance Consideration for an Operating System Transaction Manager", *IEEE Transactions on Software Engineering*, vol 15, no. 6, pp705-713, jun 1989

[155] [SR95] Shirley J., Rosenberry W.: "Microsoft RPC Programming Guide", O'Reilly&Associates.Inc. 1995

[156] [SP88] Silberschatz A., Peterson J., "Operating Systems Concepts", Addison Wesley, 1988

[157] [Sch91] Schmutz H.: "Autonomous Heterogeneous Computing. Some Open Problems", in *Operating Systems for 90s and Beyond"*, pp63-67, Springer Verlag, 1991

BUPT

[158] [Spe89a] Spector A. Z." Achieving Applications Requirements", in *Distributed Systems*, ed. Mullender S.J., ACM Press, pp19-35, 1989

[159] [Spe89b] Spector A. Z."Distributed Transaction Processing Faclities", in *Distributed Systems*, ed. Mullender S.J., ACM Press, pp191-215, 1989

[160] [Spe90] Spector A.Z.: "TABS", *Fault Taulerant Distributed Computing*, Ed. Simons, B., Spector, A., Springer-Verlag, pp115-124, 1990

[161] [Spe91] Spector A.Z.: "Some Thoughts on Systems Challenges for the 1990", in *Operating Systems of the 1990s and Beyond*. pp223-226, Springer-Verlag, 1990

[162] [SP84] Schwarz P.M., Spector A.Z.: "Synchronizing shared abstract abstract types", *ACM Transactions on Computer Systems*, vol 3, pp 223-250, 1984

[163] [SR96] Schiper A., Raynal M.: " From Group Communicatin to Trabsactions in Distributed Systems", *ACM Transactions on Computer Systems*, 2 (4), pp. 277-88, 1984

[164] [SRC84] Saltzer I. H. , Reed D. P. , Clark D. D.: End-To-End Argumenys in System Design , *ACM Transactions on Computer Systems*, 2 (4), pp. 277-88, 1984

[165] [SS85] Spector A., Schwarz P.: "Transactions: A construct for Reliable Distributed Computing", Tech. Rep. CMU-CS-85-143, 1985

[166] [Ste90] Stevens W.R.: "Unix Networking programming", Prentice Hall, 1990

[167] [Sun85] Sun Microsystems: "Remote Procedure Call Protocol Specification", Sun Microsystem Inc; 1985

[168] [SK85] Suzuki I., Kasan T.: " A Distributed Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, 3(4), 344-349, 1985

[169] [Sta89] Stalligs W.:" Handbook of computer communication standards", Howard W. Sams & Company, 1989

[170] [Svo84] Svobodova L.: "File Servers for Network - Based Distributed Systems", *Computing Surverys*, 16(4), pp 353-98, 1984

[171] [SZ89] Skarra A.H., and. Zdenik S.B.: "Concurrency control and object-oriented databases", *Object-Oriented Concepts, Databases and Applications*, ACM Press, pp.395-421, NY, 1989

[172] [THG89] Tag M.,Hebrad P., Geib J.M.: "Experience with CSA-An Object-Oriented Distributed System", *Working Conference on Decentralized Systems*, Proceedings, pp309-326, Lyon,89

[173] [TCM94] Taylor P., Cabrill V., Mock M.: "Combining Object-oriented systems and open transactions processing", *The Computer Journal*, Vol.37, No.6, pp 488-498, 1994

[174] [Tan88] Tanenbaum R.S.: "Computer Networks", Second Edition,1988

[175] [Tan93] Tanenbaum R.S.: "Modern Operating Systems", Prentice Hall 1993

[176] [Tan96] Tanenbaum R.S.: "Distributed Operating Systems", Prentice Hall 1996

[177] [Tok91] Tokoro M.: "Toward Computing Systems fot the 2000s", in *Operating Systems of the 1990s and Beyond*, pp233-239, Springer-Verlag, 1991

[178] [TR85] Tanenbaum R.S. and van Renesse R.: "Distributed Operating Systems", *ACM Computing Surveys*, 17 (4), pp. 419-470, 1985

[179] [UB87] Uyless B.: " Computer Networks: Protocol, standards and interfaces", Prentice Hall, 87

[180] [WPE83] Walker B., Popek G., English R., Kline C. and Thiel G., " The Locus Distributed Operating System", *Proceedingd of thr 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, N.N., pp 49 -70, 1983

[181] [Wei89a] Weihl, W.E." Remote Procedure Call", in *Distributed Systems*, ed. Mullender S.J., ACM Press, pp65-87, 1989

[182] [Wei89b] Weihl, W.E."Using Transactions in Distributed Systems", in *Distributed Systems*, ed. Mullender S.J.,ACM Press, pp218-237, 1989

[183] [Wei89c] Weihl, W.E."Theory of Nested Transactions", in *Distributed Systems*, ed. Mullender S.J., pp237-263, ACM Press, 1989

[184] [Wet91] Wettstein H.: "The Explainable Operating System", in *Operating Systems of the 1990s and Beyond*, Springer-Verlag, pp247-248, 1991

[185] [Wit80] Wittie L.D. and von Tilborg A.M.: "MICROS.A Distributed Object Operating System for MICRONET. A Reconfigurable Network Computer", *IEEE Transactions on Computer*, 29(12), pp 1133-44, 1980

[186] [YEK87] Yeh S., Elles, C., Ege, A. and Korth, H.: "Performance analysis of two concurrency control schemes for design environements", *Tech. Rep. STP-036-87*, MCC, Austin, Texas, 1987

[187] [Zah91] Zahorjan J., "*Operating Systems on the 1990s and Beyond*, in *Operating Systems of the 1990s and Beyond*, pp53-55, Springer-Verlag, 1991

[188] [Zay87] Zayas E.R.: "Attacking the Process Migration Bottleneck", in *Proceedings of the 11 th ACM Symposium on Operating System Principles*, Austin,Texas, pp 13-24, 1987

BUPT