"Politehnica" University of Timisoara
Computer Science and Engineering Department

A Grid Service Layer for Shared Data Programming

PhD Thesis

PhD Student: Dacian Tudor

Advisor: Prof. Dr. Vladimir Cretu

December 2009

Abstract

Grid Computing has drawn increasing attention during the last years. In spite of the tremendous efforts spent in the context of this new computing field, there is little understanding on its evolution path as well as the challenges it raises. One of the most overlooked areas, and the focus of this thesis, is the programming concept and its implications on both the application layer and the underlying system. Grid application programming solutions are currently dominated by message-passing-like constructs and distributed shared data concepts are almost not present in the grid programming landscape, although their conceptual qualities, such as higher abstraction level, suitability for highly dynamic configurations and natural fault tolerance, are very attractive reasons for building flexible and scalable grid applications. Although there are many reasons speaking for the distributed shared data model, performance and scalability related aspects are often the core arguments against them.

The objective of this work, and its major contribution, is to propose an alternative and novel solution to support efficiently the shared data programming concept for grids, by applying several ideas which to our knowledge were not put together in the grid landscape before. Other important objectives are to present a comprehensive overview and highlight the current situation in grid programming landscape, to analyze and extend some of the core qualities of a proper grid programming concept which are then used to understand the shortcomings of some of the existing solutions and consequently to define requirements from architectural point of view. The most important contribution is to design, develop and demonstrate the feasibility of a model for distributed shared data programming on the grid, suitable for systems dominated by large latency connections. The model aims to provide a more appealing programming solution based on an object oriented view and the combination of a relaxed memory consistency and type coherence. At the same level, a detailed software architecture which fulfills both the requirements and implements the defined model complements the design objectives of this work. This challenging goal has been achieved by designing and analyzing several approaches and selecting the most promising one, which has been extended by combining several ideas that have not been used together in the context of this problem domain: the model introduces the universe concept which is an abstraction of networked machines in latency proximity and defines entry consistency specification as well as specialized objects which provides additional information on data interactions.

The final objectives relate to the analysis of the designed grid service layer solution for shared data programming, which is conducted at two different levels: theoretical analysis and prototype benchmarking. The analysis is supported by defining an easy to apply and consistent evaluation methodology to highlight both behavioral and performance aspects. As the mathematical model is quite complex, the theoretical analysis aims to set performance boundaries in a stable system. In order to conduct experimental evaluations, a Java based prototype (GUN) of the grid universe has been designed and implemented. Based on a large set of experiments and their analysis, the suitability of different object types for specific interaction scenarios has been defined.

Contents

List of Figures

List of Tables

x

1 Grid Computing

In spite of all major advances in computing systems performance, traditional uniprocessor architectures may not be able to sustain the rate of realizable performance increments in the near future [1]. Concurrency has proved to be the major factor in achieving higher performance in existing systems. It is parallelism whose lack this time creates a bottleneck in system's data path and memory. By making use of parallel computing techniques, systems could make better use of existing resources, hide large communication latencies in wide distributed computing systems and provide better throughput.

Distributed and parallel applications have been used successfully in many computing domains due to the cost compelling argument. Parallel applications in engineering have been developed to solve a variety of discrete and continuous optimization problems [2]. Many of the traditional algorithms like branch-and-bound and quick-sort have been parallelized and enabled successfully on parallel architectures. Other fields of interest are represented by scientific applications where large data sets are being processed by parallel processing units. Parallel computing has also provided great support in developing commercial applications and research tools in computer systems like high processing power for cryptography and simulation algorithms.

Due to last decade's accelerated movement towards globalization and wide distribution of resources, parallel computing as presented above is not suitable to cope with the wide variety of computing demands. It is very difficult to identify and address other computing entities in a dynamic and global environment. Distributed computing has been facing many challenges ever since. Complexity and management cost for these systems has become higher and higher. Data management has raised new challenges by requesting to have the right data at the right computing location together with a higher degree of confidence through security and authentication. Although performance has been traditionally the main reason for parallel computing, challenges to performance assurance in the presence of significant load variance make it difficult to achieve. Last but not least, the system's compliance level towards users and standardization (e.g. heterogeneity) introduces another dimension in the distributed computing systems. Thanks to the networking and middleware advances, these problems have been addressed by the new evolving dimension of distributed computing, grid computing. As distributed systems have been applied to industry, the term grid has become a marketing slogan [3]. Any type of distributed file system might be called a storage grid or a file sharing application could be called a media distribution grid. According to Foster and Kesselman, grid systems must be evaluated according to their application, their business value and non trivial results rather than their architecture [4].

The term grid was introduced by the first time in 1998 by Foster and Kesselman who gave the following definition [5]: *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities"*. The definition introduces the idea of "on-demand computing" and similarities with the electric power distribution grid could be observed. Later on, Foster defined the qualities of the grid system through a three point checklist, where he mentioned also that the list still leaves room for debate. The three point checklist states that a grid system must coordinate resources that are not subject to centralized control and use standard, open, general-purpose protocols and interfaces in order to deliver nontrivial qualities of service [4]. According to Foster, a grid system should not be under centralized control, but provide decentralized resource coordination. Open standards and open source protocols must be used in order to support its grid services. All grid resources should be used in a coordinated fashion to deliver a non-trivial

quality of service. Basically, the grid definition refers to a grid system. A grid is an incarnation of a grid system, an actual working system or a concrete grid system. Most of the times the term grid is used to refer to a grid system, but the term *"the grid"* would refer to a global, ubiquitous grid in the same sense as *"the internet",* which would eventually exist in the future. We think that this stage would be the ultimate evolution point of the grid. A recent survey conducted among more than 170 grid researchers [6] shows that although there is a common understanding of the grid term, most researchers have slightly different visions about what the grid really is. One of the most debated points was the physical versus logical representation of a grid. The understanding of the relationships between grid computing, distributed computing and web based computing proved to be quite subtle, leading to quite radical opinions. For example, some think that grid computing is a combination of distributed, high-throughput and collaborative systems, while some believe that grid computing is only a concept or movement rather than a system.

By applying the above definition in distributed systems, it appears that the requirements that grid systems must provide basic functions like resource discovery, information collection and publishing, data management between resources, process management, common security mechanism underlying the above process and session recording and accounting. As all this aspects cannot be supported by the underlying operating system, all these issues are addressed in the layer on top of the operating systems, the middleware.

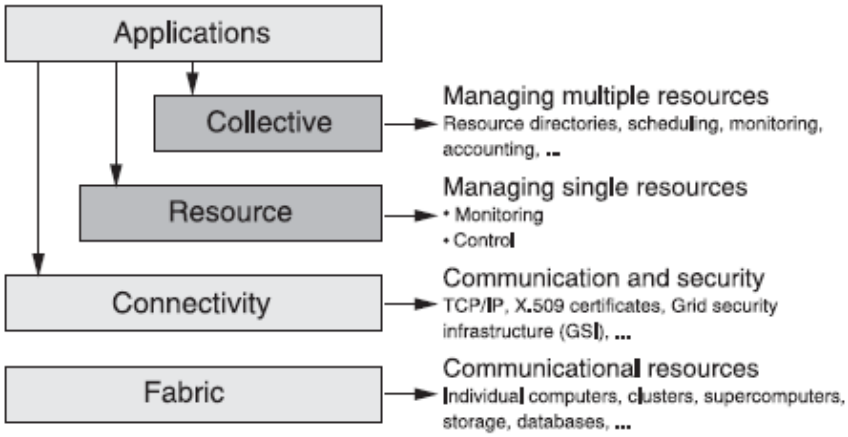

Figure 1**:** Grid Architecture as presented in [7]

A generic architecture of a grid system was introduced in [7] and was defined as a layered diagram consisting of six major categorized components as depicted in Figure 1. The bottom level component, *Fabric*, represents the hardware and software resources to be shared in the grid system. The layer above, *Connectivity*, realizes the communication between resources and provides general security services as required by the grid system, using open source secure protocols. The *Resource* layer refers to services for resource management and control for a single resource, where information and management protocols are devised in order to allow the *Collective* and *Application* layer to get resource information and request resources. The *Collective* layer consists of services that manage multiple resources where resource registries, allocation and scheduling services, monitoring services and data management services are provided. The upper layer, *Application*, represents the actual applications that are executed in the grid system. Interestingly, grid applications are not constrained to access Collective, but they could make use of *Resource* or *Connectivity* layers directly.

1.1 Grid History

We cannot talk about grid computing as a new computing field that was created as a result of a big-bang process, but rather as a slow evolution. Among the evolution path different events, ranging from technology to social and environmental aspects, have played a role in the development of nowadays grid computing domain. We believe that everything started with the demand of more computing power that initially motivated the research into high performance computing. The history of high performance computing begins in early 1940 with the Manhattan project, when the Department of Energy (DOE) of the United States aimed to develop advanced methods for some critical problems of their interests. One result of their pioneering activity was the development of the Monte Carlo method where statistical samples were used to predict behavior of large groups [3].

Another milestone in the development path of current grid systems was linking computers together in networks in 1970, when Xerox created the first Ethernet network. New developments in operating systems like timesharing systems brought the attention on the importance of putting machines to work together. Together with the Internet in the mid 1990s, two major projects brought a new direction in distributed computing. The SETI@home [8] and distributed.net projects put together an impressive number of computers in search for extraterrestrial information respectively to break encryption codes. Nowadays we are witnessing the last milestone, when our society benefits out of the high performance computing by exploring new dimensions in science, research and commerce. If consider strictly grid evolution, there are three stages in the grid system evolution.

The first stage started around 1989 in the academic world when efforts were made to link supercomputing sites by what was called *meta-computing*. In the mid 1990s projects like FAFNER [9] and I-WAY [10] provided *meta-computing* computational resources to a range of high-performance applications. The FAFNER [9] project was setup to factor RSA130 using a new numerical factoring technique on computational web servers, whereas the I-WAY [10] project aimed to integrate available US national high bandwidth networks into a high performance network linking computational units and visualization environments. FAFNER was the forerunner for SETI@home and I-WAY for Globus [11].

The second stage in grid computing evolution addressed the heterogeneity, adaptability and scalability in previous meta-computing systems. This led to quick advances in middleware technology and early efforts towards standardization. Components like grid middleware kernels, distributed object systems, resource brokers, schedulers, integrated and peer-to-peer systems represent concrete advances in grid computing technology and second generation building blocks [12]. The second generation grid computing addressed core issues like administrative hierarchy, communication services, information services, naming services, distributed file systems, resource management and reservation, system information, fault tolerance and security and authorization. These challenges led to a plethora of tools: second generation of the Globus toolkit [11], Jini [13], Condor [14], Sun Grid Engine [15], Storage Resource Broker [16], Nimrod-G [17] and many others. The end of this stage marked the introduction of grid portals like NPACI HotPage [18] and grid portal toolkits like the Grid Portal Toolkit [19] and NASA IPG [20]. Also integrated grid solutions have started to become available such as Cactus [21], Unicore [22], DataGrid [23] and WebFlow [24].

The third generation of grid systems paves the way to e-science applications and large scale service oriented architectures which promote high functional reuse. Key features like distributed collaboration and virtual organizations are some of the introduced novelties. A trend towards autonomous computing has been introduced together with self-organizing and fault tolerant grid systems.

The building blocks for such wide and complex technology have been made possible by the adoption of service oriented architectures via web service standardization technologies: SOAP [25], WSDL [26], UDDI [27] and WSFL [28]. Grid and web service technologies started to converge together with the introduction of Open Grid Service Architecture, OGSA, [29] at the Global Grid Forum, GGF [30]. OGSA aims for the integration of services across distributed and heterogeneous virtual organizations with disparate resources and relationships. These are addressed by merging the grid and web service technologies into a more generic service specification. The repeated critiques that OGSA specification was too large and cumbersome led to the specification of Web Service Resource Specification, WSRF [31]. The relationship between OGSA, WSRF and web services is illustrated in Figure 2. Similar extensions to web services are being developed for the grid in the direction of providing semantic information on grid services and information content. The semantic grid is expected to provide rich, seamless and pervasive access to globally distributed heterogeneous resources. This is the latest stage of grid development.

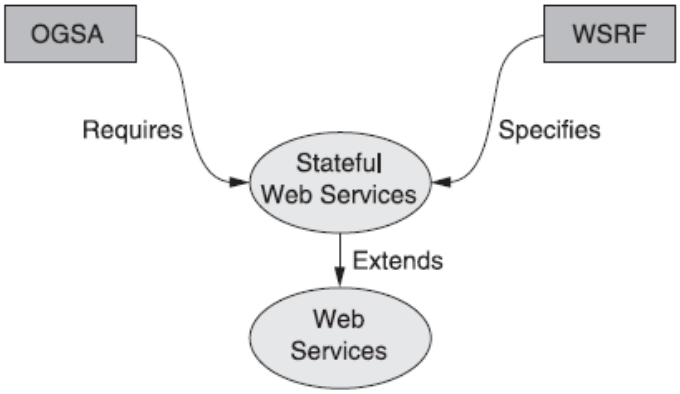

Figure 2**:** Relationship between OGSA, WSRF and Web Services

Throughout the grid evolution we could observe an increase in heterogeneous grid environments that are constructed out of large resources such as CPU and data storage. One of such systems is Grid'5000 [32] which constructed a highly configurable and controllable experimental grid. It was intended to supply 5000 processing units distributed across nine French sites connected over a 10Gb/s connection with a latency between 4 and 29ms. A similar system can be found in the US, namely the TeraGrid [33] project, which contains over 22 000 processors over nine sites. Data connections are up to 30Gb/s with a latency between 3 and 80ms that tie together a wide range of heterogeneous systems. The biggest grid system known to be put together is EGEE (Enabling Grids for E-sciencE). "*The EGEE project brings together scientists and engineers from more than 240 institutions in 45 countries world-wide to provide a seamless Grid infrastructure for e-Science that is available to scientists 24 hours-a-day. Conceived from the start as a four-year project, the second two-year phase started on 1 April 2006, and is funded by the European Commission. Expanding from originally two scientific fields, high energy physics and life sciences, EGEE now integrates applications from many other scientific fields, ranging from geology to computational chemistry. Generally, the EGEE Grid infrastructure is ideal for any scientific research especially where the time and resources needed for running the applications are considered impractical when using traditional IT infrastructures .The EGEE Grid consists of 41,000 CPU available to users 24 hours a day, 7 days a week, in addition to about 5 PB disk (5 million Gigabytes) + tape MSS of storage, and maintains 100,000 concurrent jobs. Having such resources available changes the way scientific research takes place. The end use depends on the users' needs: large storage capacity, the bandwidth*

that the infrastructure provides, or the sheer computing power available." – excerpt from the EGEE web page (http://www.eu-egee.org/).

More complex grid systems than these are expected to be deployed in the near future and pave the way towards ***"the grid"*** as the ultimate point of evolution that would be a worldwide system available to the general public as the internet is today.

1.2 Grid Applications

1.2.1 User Perspective

The last decade has seen a growing number of large-scale Grid infrastructure deployment projects including NASA's Information Power Grid (IPG) [34], DoE's Science Grid [35], NSF's TeraGrid [33] and the UK e-Science Grid [36]. NSF has many Grid activities as part of Partnerships in Advanced Computational Infrastructure (PACI) and is developing a new cyber-infrastructure initiative. Similar large-scale Grid projects are being developed in Asia and all over Europe for example, in the Netherlands, France, Italy, Ireland, Poland and Scandinavia [12]. Depending on the *application domain*, we distinguish several categories of grid applications.

Life science applications are applications that are dealing with biology, bioinformatics and genomics and are moving to grid systems due to their large computational demands and need to access and mine large data sets. Some of these projects are the Protein Data Bank project [37], myGrid project [38], the Biomedical Information Research Network, BIRN [39]. The later provides large scale simulation and analysis features and remote instrumentation capabilities.

Second category is represented by engineering applications where applications aim to take advantage of cost-effective computational resources. One of the largest engineering grid applications is owned by NASA, the NASA IPG system and represents a milestone in modern large scale engineering execution. The European counterparts are represented by the Geodise project [40] which provides engineering design knowledge repository used by Rolls-Royce among many others.

Data-oriented applications are another group of wide-spread grid applications which address the problem of operating on large data sets. An example of a data-oriented application is the Distributed Aircraft Maintenance Environment, DAME [41] which has been developed in the United Kingdom. It uses grid technology to handle gigabytes of in-flight data gathered by operational aircraft engines and to integrate maintenance, manufacturer and analysis centers.

Physical science applications are one of the pioneering applications in grid computing. Particle physics applications like GriPhyN [42], Particle Physics Data Grid [43], EU DataGrid [44] and nuclear accelerator analysis projects at CERN have been an early proof of grid technology applied to large data flow and highly parallelized applications.

Last but not least, commercial applications have been started to emerge in the context of enterprise computing and they are addressing problem domains like automation, security, utility computing, computing cost reduction, disaster recovery etc. They have been starting to benefit from the collaborative approach in e-Science that proved to be a modern approach in sharing collective wisdom. An overview of grid application distribution by the topics addressed has been presented in the GridCoord report [45] and reproduced in Figure 3.

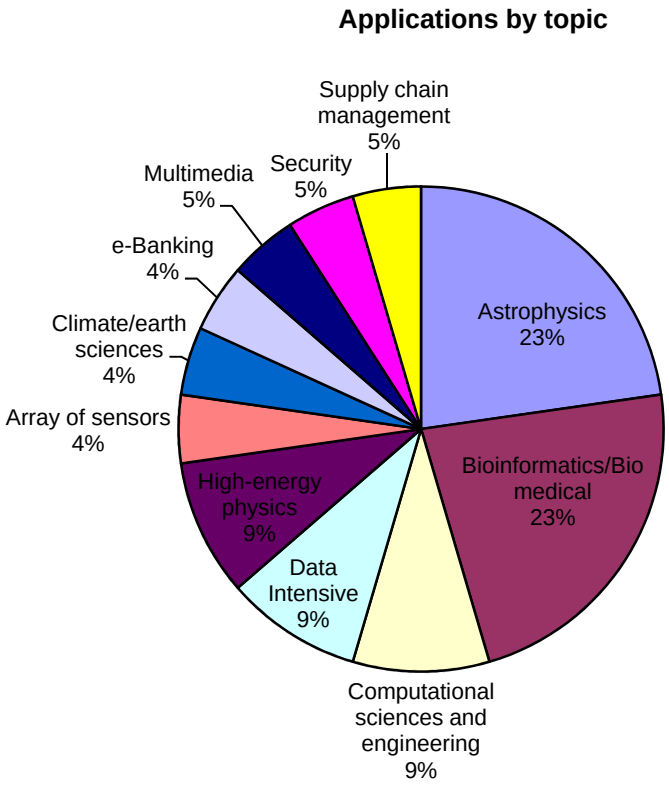

Figure 3: GridCoord Application Distribution by Topic

1.2.2 System Perspective

Considering the application architecture and factors like data locality and task or service interaction patterns, grid applications could be divided in four categories: loosely coupled applications, pipelined applications, tightly synchronized applications and widely distributed applications.

Loosely coupled applications are made out of small tasks with typically small memory, data and communications demands, but very compute demanding. They could be running in large cluster environments with large latency and low bandwidth characteristics. One of these applications is SETI@home [8], where input data decomposition has been applied before submitting the data set for processing. *Pipelined applications* are typically processing streams of data like multimedia or other kind of real time broadcasted data. These applications have higher memory and data demands than loosely coupled applications and some of them impose hard constraints in terms of data path delivery. Tasks are typically organized as coarse grained parallel tasks and require more task communications. Applications in this category are dealing with real-time data acquisition and processing of large data sets, acquired for example via satellites or remote sensors which might be located outside of the data processing units. One of these applications is the Biomedical Information Research Network [39].

Tightly synchronized applications are data-intensive applications that have the same requirements as the previous ones, but in addition they are exhibiting intensive synchronization on the communication infrastructure by intensive inter-task communication patterns. These applications are running in high performance computing systems which provide enough latency for intensive

communications. Modeling applications such as climate, physics, biology or aeronautics fall in this category. *Widely distributed* applications are data oriented applications that are performing a series of operations on wide distributed data sets. They have little demands for memory, data and computation resources, but need to work on a heterogeneous and widely distributed grid infrastructure and access data owned by different organizations. Biomedical applications that are searching and updating large data sets are part of this category of applications.

Summarizing the application view from the system perspective, grid systems could be divided into three categories: *computational, data and service grids*. *Computational grids* are aggregating computing resources and provided a total computing power higher than the power of any individual part. Depending of their usage, computational grids could be further divided into high throughput grids and supercomputing grids. While supercomputing implies that multiple jobs are run in parallel to reduce the completion time of a job, high throughput grids are focusing to improve the completion rate of a stream of jobs, by trying to run as many jobs as possible in a given time unit. The *data grids* focus on providing and managing data sets in distributed grid environments. The latter category, *service grids* is an emerging concept inspired from service oriented architectures and basically addresses on-demand, collaborative and workflow computing. Collaborative and on-demand computing provide services that are not provided by any single machine, by dynamically aggregating different resources and services. Interactions between users and machines are thus possible through dynamic virtual services.

1.2.3 Grid Perspective

Any grid-enabled application requires some services of the underlying grid infrastructure in order to perform properly. There are other applications such as monolithic legacy application which do not require any special execution support which can be simply run transparently on a grid system. However, the grid system is still required to perform some tasks in order to provide the necessary execution context for the target application. As the grid is continuously evolving and is progressively defined by bodies such as the Global Grid Forum, it is important to identify the generic grid application types as well as the grid specific operations that are required during their deployment and execution. On the above types of grid systems one can develop several types of applications. Based on the primary focus of the application, the following taxonomy can be devised: community-centric, data-centric, computation-centric and interaction-centric applications.

First category of grid applications are *community-centric* applications that aim to bring together people or communities by enabling different kinds of collaborative interactions and workflows. Large scale interactive video conferencing systems and scientific collaborative systems are two examples of community-centric applications where workflow management plays a key role among distributed resource.

Data-centric applications are data oriented applications that supply large scale distributed data from various data providing sources like sensors, satellites or data processing units. Most of the time, they provide specialized data access to different kinds of data. Typically, data-centric applications are operating on large scale distributed data repositories that are commonly found in scientific and environmental applications. These kinds of applications have been grown very much during the grid development stages and their growth rate is expected to continue on the same rate.

Computation-centric applications are focusing on achieving high-computational throughput that is required in many simulation and prediction application such as climate modeling, avionics, economic forecasting, and industrial modeling. They represent most legacy high performance computing applications that see the grid as a computational resource pool for their intensive computations. It is expected that these applications will move from parallel processing to grid processing, in order to overcome the resource limitation of parallel computers.

Last but not least, *interaction-centric* applications refer to time critical applications where user interaction happens during a decision-making process that most of the times involve visualization. It is required that the application provides a quick response, thus issues like load balancing, resource reservation, performance assuring are key points to be addressed in this case. Currently, the interaction level is reduced to visualization or simple user interaction, but event standardization could automate the life cycle of these applications and increase their throughput which is limited by the interaction process.

The grid operation taxonomy can be devised in respect of grid operations types: basic operations, information and interaction operations and compound operations [12]. *Basic operations* are building blocks for any grid job execution and consist of resource selection, job initialization and data transfer operations. In a grid environment which consists of many different types of resources, one or more resources have to be selected to carry through the designed task. Resource selection is done based on resource registration, locating and monitoring. Job initialization refers to provide the proper execution context and credential verification plus supplying the necessary input data (known as data staging). Data operations are typically basic data transfer operations such as file copying on a grid level. Based on the simplified grid data transfer model, advanced data operations could be provided such as reliable data transfer, data replication and searching.

Information and interaction operations are a great help in managing and collecting different kinds of information in a grid system. Information can refer to resources (CPU, storage), software, users, data etc. A special class of information is application monitoring which is a fundamental requirement for job scheduling and resource reservation operations. Information can be delivered by either push or pull mechanisms (interaction). The pull approach is simply user or application driven, whereas push involved reliable notification mechanisms to the user or different grid components.

Compound operations are built on the above mentioned operations and provide new and versatile features in the grid environment. Combining resource selection, information retrieval and data transfer results in a new *migration operation.* Application migration implies changing the execution location of a job, by transferring its state and data to a new execution node. Thus, the grid execution framework could provide new resources where a job can be finished earlier as in its previous location. Spawning is another compound operation that involves creating sub-jobs of a main job and spreading them on the grid. As processes might be executed recursively, a hierarchy of complex grid workflows can be created. Such an operation requires resource acquisition, data transfer from the main job or process to the other resources, initialization and collecting the results from the sub-jobs. Combining job migration and job spawning leads to task farming. Task farming is a typical case of parameter search solution where a main master process spawns processes on the grid that search a subset of the possible combinations. The main process spawns new processes until its task is completed and the search space is exhausted or the solution is found.

1.3 Grid Programming Models

During the last two decades we have witnessed a massive movement from traditional client-server computing to distributed, network based computing and lately service oriented architectures. Grid programming introduces many challenges that were not addressed previously in sequential computing, parallel and distributed computing. Basically, grid programming orchestrates interactions between resources, services, shared or distributed data structures in a heterogeneous and dynamic environment. Most of the current grid applications mainly consist of job submissions, or they are parallel or distributed applications that have been ported on top of grid middleware and transformed in grid enabled applications. Next generation of grid applications, that are going to unleash the envisaged computing potential, involve heterogeneous and dynamic interaction between different types of systems and resources with run-time changes in topology and configuration. Dynamism and disparate resources are core features that grid systems and programming methodologies must address, instead of putting the burden on the application side.

When referring to a programming model, we refer to the conceptual computation orchestration technique and not necessarily to a programming language. A programming model can be presented in several forms such as an application programming interface (API), a tool with external functionality or a conceptual model. In the grid context, most successful programming models must enable high performance and flexible resource composition and management. At the same time, a programming model is directly influencing the entire software life cycle: design, implementation, debugging, operation, maintenance [12]. There are several issues that grid programming paradigms must address in the context of future grid systems. These include wide area scalability, latency and bandwidth hierarchy, fault tolerance, automatic management capabilities. It is clearly desired that grid programming paradigms must have certain programming properties that support building high quality and efficient grid codes. Next, we present the grid programming properties and then the main distributed computing programming models.

1.3.1 Grid Programming Properties

Programming properties have been introduced since the time of parallel computing. Extending these properties for grid systems is one important step towards providing the means to design and develop efficient grid codes. Ideally these characteristics have to be addressed by the programming tools and paradigms and not to be explicitly programmed by the application programmer. Based on [46] there are seven important properties of grid programming models. In addition, we extend the programming properties with a new one that we see as a true challenge that needs to be addressed in grid programming models.

Usability. Grid programming tools must support various types of programming concepts and paradigms from local computing to large scale high performance computing. There should be no constraints in building program codes that are targeted to a specific architecture so that different development paths are followed depending on the system's requirements and architecture.

Dynamic and Heterogeneous Configuration. Grid applications are running in dynamic and heterogeneous environments that change frequently due to machine availability, new connection paths, dynamic communication latencies due to connection changes, new resource availability or potential failures. In order to have an efficient programming paradigm, configuration and architectural details must

be ignored by the programmer. These must be addressed internally by either the framework or additional components.

Portability. Portability is not a new topic and is best captured by the sentence “write once, run anywhere”. For grid systems, portability is similar to supporting programs to be run independently of the underlying architecture. Portability and architecture independence is vital to support dynamic and heterogeneous configurations.

Interoperability. Grid systems are based on open standards and protocols. Open protocols assure a greater level of inter-operability. Without using open standards and protocols the possibility to extend grid architectures is reduced. Protocols, services and interfaces might expose programming models that must be interoperable as well.

Reliable performance. One of the motivations for grid systems is to offer tremendous amount of cost effective computing power by harnessing unused distributed resources in an efficient and uniform way. Thus, one of the most demanded requests for grids is high quality reliable computing power. In some special cases predictive performance, known in deterministic models, is a demand that the grid must deliver with high quality and accuracy. Grid programming tools must not limit the performance of the developed application. In other words, the performance bottleneck should not be the programming tools or paradigms. Besides reliable performance, another key aspect is that of performance portability.

Reliability and Fault Tolerance. Previously reliability and fault tolerance were addressed in the developed solution. In case of grid systems, this is not applicable anymore as grids aim to expose higher level functionality with advanced management support. Still, errors and failures must be notified, but built-in run-time mechanisms must be present to allow automatically error detection and correction.

Security and Privacy. As the grid spans between virtual organizations with possible different security policies, security issues, rights management and privacy have been a major concern. As grid codes are running across different administrative domains, it is very important that security be part of grid programming tools. Basically this is solved by the grid middleware by delegating credentials. However, delegation chain might be the weak link if not handled properly by every indirection level.

Uncertainty is in our view a final challenge that grid systems must address that appears due to several factors. Most obvious factor is the dynamic and unpredictable environment which might change not only in structure, but also in behavior (e.g. due to service upgrades). Failures play an important role too in the landscape of uncertainty as their chances increase as the grid expands (its scale increases). Last but not least, incomplete knowledge of the global system state, which is a common characteristic in distributed asynchronous systems, contributes to the increase of system uncertainty too.

1.3.2 Client-Server Model

The client-server model is the simplest distributed computing model, where a node called server offers services that are requested by clients. In this model, the client is always a consumer whereas the server is a provider. The model is demand driven as clients are actively requesting server services while servers are passively waiting for requests. The interaction between clients and servers could be stateless or statefull, but there is no consent of state management which is highly application dependent. Complex applications could be developed using this model by organizing clients to access concurrently multiple servers and servers to serve multiple clients in parallel.

This centralized setup increases the dependencies between clients and servers and might be a bottleneck in case the server cannot process enough requests simultaneously. A workaround to alleviate the server bottleneck is to provide a layered architecture containing multiple servers organized in different levels. Such solution was applied for example to the DNS servers. DNS like systems have the advantage that they are easy to manage and identify, but have the single point of failure which makes them not suitable for highly available systems.

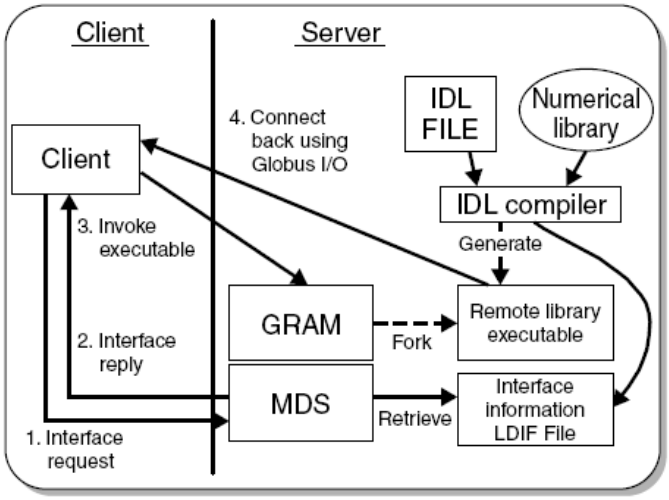

Figure 4: GridRPC Architecture

In this category we consider Remote Procedure Call (RPC) and Remote Method Invocation (RMI) models which involve calling a procedure or a remote object's method located at a remote server. The semantics associated with the call is determined at design time, thus no explicit parameter marshaling is required. At the grid level, the GridRPC [47], an extension of the RPC model, provides standard RPC operations and specific grid operations like dynamic resource discovery and scheduling, security via X.509 certificates and fault tolerance through automated check-pointing, rollback and retry features. Different implementations of GridRPC like Ninf [48] or NetSolve [49] which provide powerful client-server programming frameworks based on remote procedure call. In such a system, applications are written as task-parallel programs that invoke a remote executable located on the grid. The remote executable contains computation code that is executed asynchronously as a result of a RPC call. A generic view on the GridRPC architecture is depicted in Figure 4 where GRAM represents the server component of the RPC and MDS represents a registry containing registered GridRPC components.

One of the drawbacks of RPC-like solutions is the assumption about the common knowledge of procedure names and identifiers. More, it defines and freezes the syntax and semantics of the remote interface a priori, at compile time. The model is pretty static and it does not address dynamism, uncertainty or security. It also puts a burden at the application level by forcing the application to handle reliable message delivery and providing basic mechanisms for failure management. RPC-like approaches address heterogeneity by using neutral description languages for the interfaces (e.g. IDL). At this time, there is a proposal to enhance the RMI model for the grid which could significantly improve the unnecessary communication overhead of method compositions on the grid [50]. Web services fall in the same client-server model, the only difference being in the invocation details, where parameters are marshaled in language neutral XML representation, ensuring thus platform and language independence

and the ability to build loosely coupled systems. Of course, the price for flexibility and loosely coupling is paid by the higher costs of XML representation conversions.

1.3.3 Peer-to-Peer Model

Peer-to-peer systems (P2P) are decentralized distributed systems, by removing the roles of clients and servers and replacing them with a generic peer component which exposes uniform services in the entire peer-to-peer system. A peer-to-peer node performs both roles of client and server and cannot be distinguished from other nodes. The role of the peer to initiate or respond to other peers depends on what role fits better to the use of computing resources such as CPU cycles, space or network bandwidth. In most peer-to-peer models, peers have similar computing power, have same capabilities and can handle multiple concurrent interactions. Peer-to-peer applications have been used to provide file sharing, web caching, distributed informative systems and other kind of services that are living in a large and dynamic distributed environment. These applications are highly efficient when the data they are operating on is immutable.

Peer-to-peer middleware such as Pastry [51], Tapestry [52] or Chord [53] are recent technologies that have provided a great boost in the development of peer-to-peer applications. The peer-to-peer middleware is making use of current naming, routing, and replication and security techniques in new ways to build a reliable resource sharing layer over an unreliable and untrusted collection of computers. A family of protocols designed for the Java language and P2P computing is the JXTA framework [54]. The JXTA framework defines the P2P communication through XML messages which provides support so that peers can form self-organized and self-configured groups without any centralized management infrastructure. Peers can make advertisements and can communicate and route messages by making use of JXTA protocols which are shielding the complex and dynamic underlying communication infrastructure. According to [55], JXTA can be successfully used to build Java-enabled grids.

1.3.4 Workflow Model

A workflow is an automation of a process, where information or tasks are passed from one participant to another where an action is to be applied, according to a set of predefined rules. The concept of the workflow model relies on service oriented architectures and collaborative systems. The service oriented architecture is a collection of message oriented services defined in terms of messages exchanged between providers and requestors. Services are description oriented and are described using standardized meta-information (e.g. in XML dialect). Such services are implementation independent, network use oriented and have typically a small number of operations with large and complex messages. Service semantics is separately documented as the service specification does not mandate any semantics.

Most popular and promising service oriented architectures are based on web services. These could easily be organized in workflows. There are many web service composition languages like PDL, XPDL, BPSS, EDOC, BPML or BPEL4WS which provide web service workflow models that are widely used in industry, commerce and research domains [56]. Composition and flow specification for languages for web and grid services include BPEL [57], Grid Services Flow Language, GSFL [58], Web Services Flow Language, WSFL [59], XLANG [60] or Web Services Choreography Interface, WSCI [61]. Scientific workflow models typically require large data volumes, advanced workflow monitoring and control support,

dynamic configuration and hierarchical composition. Such requirements make them perfectly suitable for grid systems and standardization work is an ongoing activity [58].

1.3.5 Message Passing Model

The message passing model refers to processes running in separate address spaces that are exchanging information between each other via messages. Typical roles are publisher and subscriber, where publishers are actively publishing a stream of data to a communication channel and subscribers are passively receiving data through the subscribed channel. Unlike the client-server model, the publisher is active and not passive like the server and the subscriber is passive unlike the active client.

Message passing specifications like MPI [62] define inter-process primitives that shield programmers from communication issues due to complex network protocols and heterogeneous platforms. It enables communication using primitives such as *send* and *receive* by explicit data copying from one process to another. Data access and synchronization control is left to the applications level, as in all previous models. This model is also known as *share nothing*. One of the main advantages of this model is performance, the only limiting factor being the communication medium congestion. On the other side, this model is shifting much responsibility on the application side where the application has to manage application specific communication protocols, must address dynamic configurations and built support for fault tolerance. Other strengths of this model consist of high portability and user control. In other words, message passing programs are highly portable and they give the programmer total control over the data transfers between processes.

Although message passing tools focus on performance, they do not support heterogeneity, dynamism or uncertainty. Further, they assume that all processes are trusted and they do not address security. MPICH-G2 [63], a grid enabled implementation of MPI hides heterogeneity using services provided by the grid middleware. It makes use of Globus [11] services to couple together multiple machines. It supports advanced data transformation routines and multi-protocol communication. In terms of dynamism support, MPI-2 [64] specification adds dynamic process creation and runtime modification of the processing sets. A refinement of the message-passing model is represented by the register and notify model, where consumers are registering to a service provider for a lease period. Providers are notifying consumers with service data during the valid lease period. The roles are similar to the publisher and subscriber where the provider is active and the consumer is passive. Such model is implemented in the COM+ [65] and Jini [13] technologies.

1.3.6 Distributed Shared Data Model

The distributed shared data model exposes a global shared memory space that is shared among processes on loosely coupled processors and which creates the illusion of a single large memory space. Basically, it integrates local memory in a networking environment into a single entity, shared by multiple processes located at different sites. The programmer has thus the illusion of a large address space. This concept emerged from the hardware shared memory used between multi-processors and was extended on loosely coupled processors, which is also known under the term of distributed virtual shared memory.

One of the most well known specifications for distributed shared memory model (DSM) is the tuplespace model which was introduced in 1982 in the context of the programming language called Linda [66]. Tuplespaces are abstract communication and computation environments that form the basis of the

Linda communication model. The Linda [66] programming language uses shared object spaces, known as tuple spaces, which are accessed by standardized methods. Communication is done by placing and removing tuples from the shared memory space. A similar implementation has been provided by Java Spaces [67] in the Java programming language that was introduced by Sun Microsystems, where a Java API is provided for the tuplespace model and language specific extensions for richer typing, object orientation, subtype matching and transactional support. TSpaces project [68] represents another Java implementation of the tuplespace model which adds database capabilities and is oriented more on data repositories than global communication. Implementations like Linda [66] and Java Spaces [67], where communication is done by placing and removing objects in a global space, make available higher level abstractions that are easier to use and program. However, due to the new interaction approach, which does not happen explicitly anymore, there is a slight thinking shift needed while using the object space approach. This model seemed very appealing to its initiators and was thought as of a concept that could support any computational model [69].

The main advantages of this model are better usability, support for dynamic configurations and fault tolerance as it naturally addresses all of them. Performance and interoperability have been seen always as weak points; perhaps this is why the model is not as popular as the message passing approach.

1.3.7 Parallel Programming Models and Technologies

Message passing and shared data models represent the most important concepts of parallel programming models. Parallel programs can be divided in two categories, depending on the aspect they parallelize, namely the execution of the same task on different data elements (data parallelism) or the concurrent execution of different tasks on the same data element (task parallelism). Most parallel and grid programs are data parallel in nature. One of the reasons for this characteristic is that the concurrency level that can be algorithmically obtained via data parallelism is higher than the one represented by task parallelism. In other words, it is easier to elaborate an algorithm and apply it on different data, rather than devise the same number of algorithms as data structures and apply them on a single data item. However, task parallelism is quite important in the context of distributed computing, where distributed components run different codes on potentially the same data items, but on different locations. ***In the context of grid computing, we expect to witness grid codes that are structured as a task parallel composition of data parallel components.***

From the programming perspective, parallelism can be expressed either explicitly or implicitly. Normally, an explicit parallel program contains programmer written statements that define concurrent activities such as control threads. On the other side, implicit parallelism relies on higher level constructs and specifications of program behavior, which are translated by a compiler or run-time environment into parallel activities. The flexibility and total control over the parallel activities that is a core characteristic of explicit models is not present anymore in case of implicit models, where the programming activity is highly simplified, but the user does not have complete control over the parallel activities.

There are a plethora of technologies that supply an implementation of parallel models, both message passing and distributed shared data. Some of them have been already referred in the above sections, but there are many others that were not mentioned. We highlight briefly some of the most important technologies for parallel programming that are commonly used in nowadays applications.

MPI [62] is regarded as the technology of choice when it comes to constructing scalable parallel applications with a high portability degree. There are a plethora of MPI libraries and bindings to programming languages like C, C++, Java and Fortran. MPI is regarded as an efficient communication solution that puts some extra burden on the developer in order to orchestrate data exchanges between remote processes. ***PVM*** [70] is another implementation of message passing model and represents one of the predecessors of MPI libraries. One of the major design goals was portability which turned out to sacrifice performance. The evolution path of MPI standards (MPI-1 and MPI-2) added dynamics and portability and provided better performance than PVM implementations. As a result the PVM approach began to suffer in terms of popularity. ***Parallelizing compilers*** are an appealing tool for generating parallel codes automatically out of high level constructs. Due to the complexity of parallel compiler construction, they had success primarily on shared memory architectures with a small number of processors. In general the performance obtained using parallel compilers is small if no information is provided by the user and the entire code generation is done automatically. However, the situation is greatly improved when user provided annotations are supplied into the code. ***OpenMP*** [71] is a parallel programming solution that makes use of a parallel execution library and a parallel compiler in order to supply a solution for shared data programming. It is intended for shared data architectures and provides bindings to C, C++ or Fortran. OpenMP exploits loop parallelism in both fine and coarse grain parallel activities and preserves sequential execution semantics. ***High Performance Fortran*** [72] is an extension of the Fortran language with a set of directives and new language constructs to provide a data parallel programming model. Different to OpenMP, High Performance Fortran focuses on user support for data distribution in order to support a high performance execution on heterogeneous machines, especially on shared memory architectures. Solutions for implicit data parallelism include libraries like ***POOMA*** [73] and ***HPC++*** [74] that abstract parallel operations. Both solutions rely on standard object oriented technologies that define classes in order to encapsulate parallelism. One of their major advantages is code clarity and clearly defined abstractions. The down side of these approaches is opacity, because debugging and library extensions are very complex tasks.

The importance of parallel programming technologies can be considered in the context of evolution of parallel systems towards large scale complex and hybrid entities. Architectural advances that translate into higher complexities restrict their availability for the general programmer and require more and more the exclusive expertise of a domain expert. We believe that this natural evolution aspect has high chances to slow down the development of grid applications. As a result, one must rely on flexible, complex and at the same time easy to use programming models suitable for the development of grid applications. Considering the impressive technologies and concepts that are currently involved in grid computing, we expect a slow convergence process towards mature and generally accepted programming models for grid programming.

1.4 Grid Middleware

Grid middleware is a software layer built on top of operating systems services that provides a series of cooperating services and interfaces in order to offer transparent and seamless access to grid resources. Some of the most important grid middleware solutions are briefly described in the following sections. A grid middleware is used in order to construct grid systems like (listed in [75]):

- EGEE: Enabling Grids for E-Science in Europe (http://public.eu-egee.org/): An ambitious Grid project that brings together scientists and engineers from more than 240 institutions in 45

countries world-wide to provide a seamless Grid infrastructure for e-Science that is available to scientists 24 hours-a-day. EGEE will also be responsible for providing the awesome computational power required by the LHC described above.

- NEESit (http://it.nees.org/): Provides an extensive infrastructure for the NEES (Network for Earthquake Engineering Simulation) Collaboratory by linking together earthquake research centers in the US.
- TeraGrid (http://www.teragrid.org/): A Grid system providing a powerful infrastructure for open scientific research. As of 2004, TeraGrid had 20 teraflops of computing power and 1 petabyte of distributed storage.
- Access Grid (http://www.accessgrid.org/): A Grid system used for “large-scale distributed meetings, collaborative work sessions, seminars, lectures, tutorials, and training”.
- eDiaMoND (http://www.ediamond.ox.ac.uk/): The eDiaMoND project is an example of how Grid computing can be used for e-Health. This project “pools and distributes information on breast cancer treatment, enables early screening and diagnosis, and provides medical professionals with tools and information to treat the disease”.

1.4.1 Globus Toolkit

The Globus Toolkit [11] is a grid middleware mainly developed at the University of Chicago and represents a community-based, open-architecture, open-source set of services and software libraries that supports grids and grid applications. It provides a collection of components that supply a ready to use framework for building collaborative distributed applications. The focus is to provide tools and protocols that address heterogeneity in large scale distributed systems and to provide reference implementations to the latest protocols adopted by standardization bodies like IETF, W3C, OASIS or OGF.

The toolkit includes software components provided as web services or non web services for the following grid problem domains: security, information infrastructure, resource management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications. The current Globus Toolkit (version 4.2.1, version 5.0 being announced for December 2009) has reached the forth evolution stage. Its main modules are depicted with different colors in Figure 5. The Common Runtime components provide core tools for building web-service and non-web-service services. The Security module provides secure communication means in the grid and defines the Grid Security Infrastructure (GSI). Data management module provides protocols and tools to manage large data sets like the reliable FTP and GridFTP protocols. Information services components comprise monitoring and discovery services (MDS) that enable uniform resource location and monitoring in a virtual environment. Last but not least, the execution management module provides the management services for job execution, scheduling, coordination and monitoring.

In the last two releases (starting with 4.00 and reaching 4.21 at the time of writing this thesis), Globus has focused to provide web service interfaces for mainly all its components and has been redesigned to enhance scalability, modularity, performance and usability. At the same time it leverages existing web service standards like WS-Addressing, WS-Security and WS-I Basic profiles and accommodates further standards like WS-Resource Framework and WS-Notification. The architecture of its web service runtime is depicted in Figure 6.

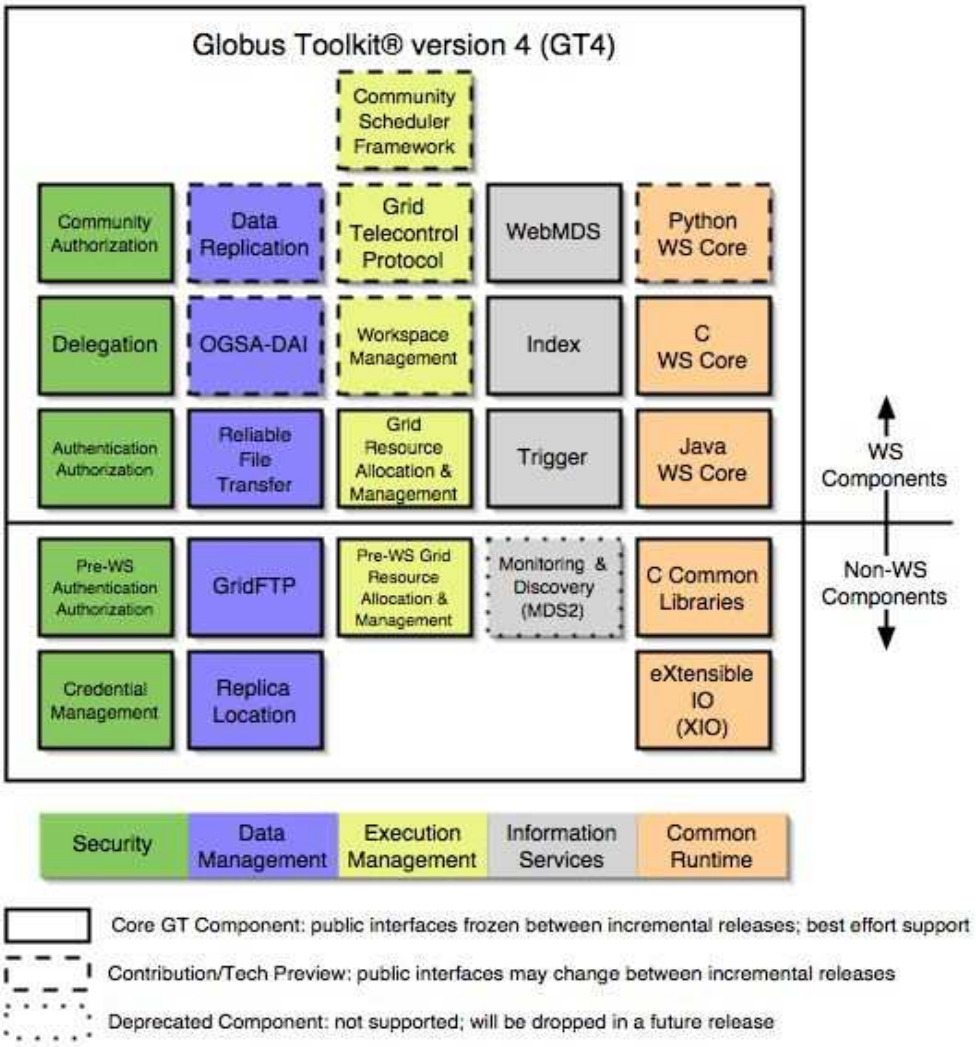

Figure 5: Globus Toolkit Components, according to www.globus.org

Grid security services provide support to control the access to grid resources. It allows multi-user collaborations through federated mutually trusted services. Globus users are allowed to set up dynamic trust domains supporting thus user specific resources that are working together to a common goal. The entire security module is built on top of web service standards. Credential management is done by the single sign-on proxy called MyProxy and Community Authorization Service. Data management components provide functionality for moving large data sets through the GridFTP protocol. Data staging is integrated into the execution and resource allocation framework, GRAM (Globus Resource Allocation Manager). Globus provides data access mechanisms by providing replication catalogs via the Resource Location Service (RLS). Efficient and reliable data replication is provided by the Data Replication Service (DRS). Globus provides mechanisms and interfaces for data access such as GridFTP for file access and OGSA-DAI [76] for any kind of data (database, files, memory).

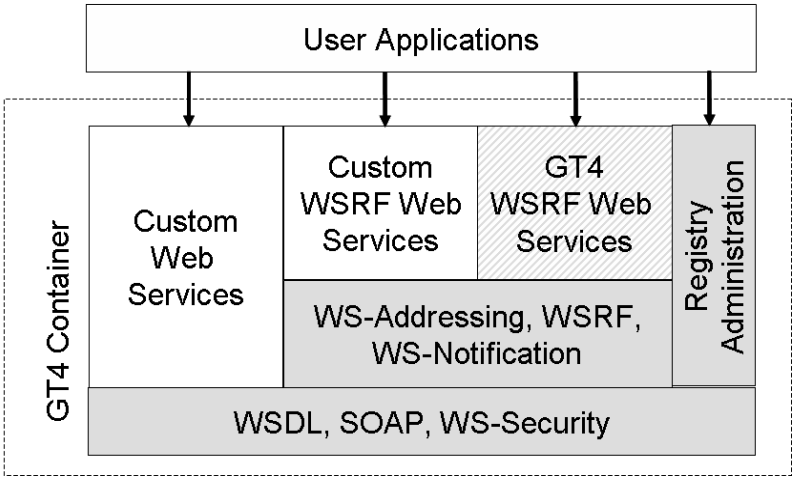

Figure 6: Globus Web Service Architecture, according to www.globus.org

The execution management component (GRAM) provides an interface for process execution that comprises setting up the execution environment, data staging, monitoring and managing the process life cycle and returning the results and clean-up. It supplies the basis for application provisioning and exposes a web service interface for scheduler integration such as Condor, LSF, and PBS. The architecture of the GRAM module is depicted in Figure 7. Monitoring and discovery components provide basic means for monitoring and discovering all published web services. Globus provides a framework for hierarchically collecting information about its services, meaning that all Globus services such as GridFTP and GRAM are discoverable. Information is indexed and service changes are sent as notifications to the subscribers via the WS-Notification interface.

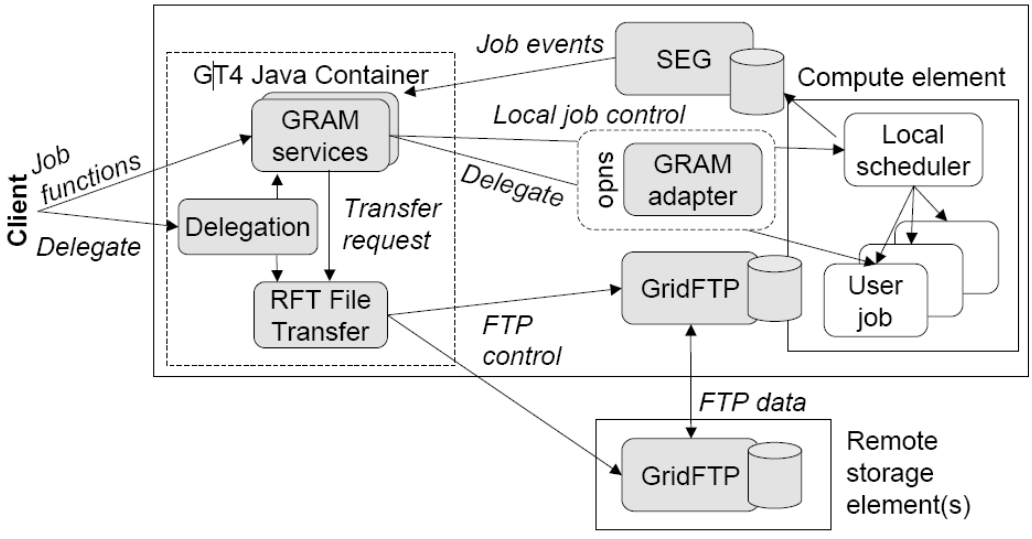

Figure 7: Globus GRAM Architecture, according to www.globus.org

The execution management component (GRAM) provides an interface for process execution that comprises setting up the execution environment, data staging, monitoring and managing the process life cycle and returning the results and clean-up. It supplies the basis for application provisioning and exposes a web service interface for scheduler integration such as Condor, LSF, and PBS. The architecture of the GRAM module is depicted in Figure 7. Monitoring and discovery components provide

basic means for monitoring and discovering all published web services. Globus provides a framework for hierarchically collecting information about its services, meaning that all Globus services such as GridFTP and GRAM are discoverable. Information is indexed and service changes are sent as notifications to the subscribers via the WS-Notification interface.

1.4.2 gLite and OMII

gLite [77] is EGEE's next generation grid middleware developed in the context of the EGEE-II project. This project is a large scale project that brings together scientists and engineers from more than 240 institutions in 45 countries world-wide to provide a seamless Grid infrastructure for e-Science that is available to scientists 24 hours-a-day, by establishing a WSRF and OGSA compliant service oriented architecture. The gLite architecture depicted in Figure 8 follows the service oriented architecture pattern and provides higher level service decomposition as following: security services, grid access services, information and monitoring services, job management and data services. At time of this writing, EGEE Grid consisted of 41,000 CPU available to users 24 hours a day, 7 days a week, in addition to about 5 PB disk (5 million Gigabytes) + tape MSS of storage, and maintains 100,000 concurrent jobs. Having such resources available changes the way scientific research takes place. The end use depends on the users' needs: large storage capacity, the bandwidth that the infrastructure provides, or the sheer computing power available.

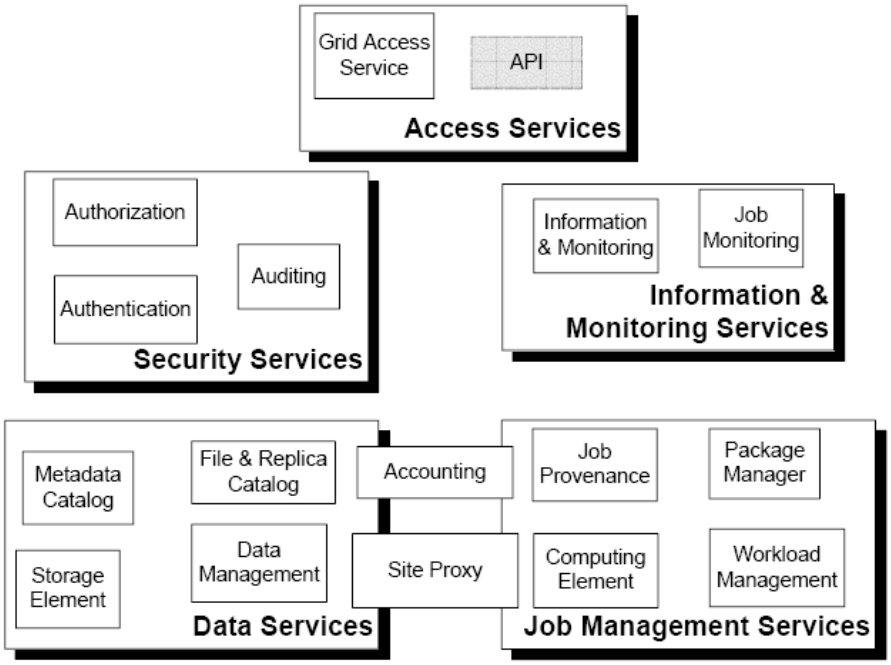

Figure 8: gLite Architecture

Security services contain the Authentication, Authorization, and Auditing services which enable entity identification such as users, systems, and services. It controls access to services and resources and provide security information and data confidentiality support. *Grid Access Service* provides a common interface that provides users access to grid services. The access service manages the life-cycle of the grid services available to a user, according to his privileges. *Information and Monitoring Services* provide a mechanism to publish monitoring information. The information and monitoring system can be

used directly to publish, for example, information concerning the resources on the Grid. Specialized services, such as the Job Monitoring Service, are built on top of the basic informative service. The system relies on registering the location of publishers of information and what subset of the total information they are publishing. This allows consumers to issue queries to the information system while not having to know where the information was published.

Job Management Services are the main services related to job management and execution. They contain resource management, workload management, accounting, job provenance, and package manager services. Accounting is a special case as it considers not only computing, but also storage and network resources. These services communicate with each other as the job request progresses through the system, so that a consistent view of the status of the job is maintained. *Data Services* consists of three main service groups that relate to data and file access: Storage Element, Catalog Services and Data Management. Closely related to the data services are the security-related services and the Package Manager. Data management services are operating on the file level, but extensions towards data sets and catalogs have been considered. One of the novelties of these services is the provisioning of data scheduling that exposes interfaces for data placement in the grid environment.

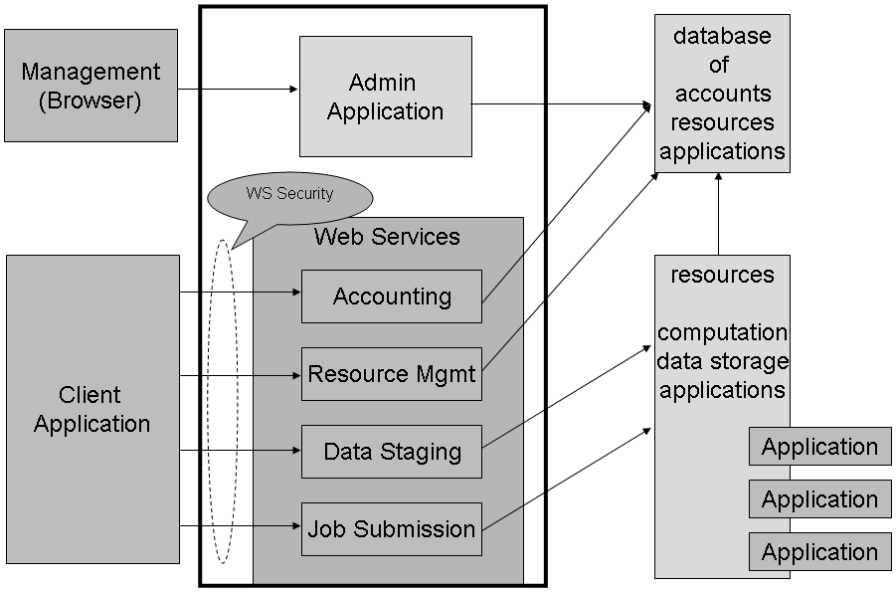

Figure 9: OMII Architecture

Open Middleware Infrastructure Institute (OMII) [78] is an UK based initiative to develop a web service oriented infrastructure for building grid systems. It is an open source project which adds security to web service architecture and uses only existing open source software (e.g. Axis, Tomcat) and standards. It offers file based job submission and tracking tools in a secure environment built on top of classical web services. Its exposed services are pure web services for accounting, resource management, data staging and job submission. OMII is mainly focused toward e-Science solutions and aims to provide a software repository of grid components from e-Science projects, by re-engineering and refining the components sourced from the scientific community. The architecture of the OMII is presented in Figure 9. First version of the OMII middleware used the Tomcat application container and AXIS SOAP engine with WS-Security extensions. It provided only a job submission interface via the Process Based Access Control (PBAC) module. Basic services that were offered were accounting services, resource management, data staging and job submission.

1.5 Grid Shared Data Dilemma

Although the number of networked machines has been constantly increased, the number of new distributed applications is still much lower. One of the obvious reasons for such a slower growth factor is the degree of difficulty in creating and maintaining wide distributed applications. Some of the core issues that are faced by distributed applications are due to latencies, synchronization and partial failures. Most of the times, communication still takes place over large latency connections and requires a much longer time relative to the processing speed. Only on dedicated grids ideal conditions can hold during the entire application lifetime. Next, the increasing heterogeneity and the greater difficulty to replace large spread legacy systems impose an important break on grid application development. Last but not least, the emergence of mobile computing raises complex questions on application deployment and performance assurance in various contexts. ***One of the answers we see to these challenges is in the grid programming model. This is the major focus of this work.***

As shown in previous sections, grid systems consist of both computational and data resources. According to the grid definition introduced in the beginning of this chapter, grid systems must provide transparent and standardized support to their resources including transparent access to application data. If simply considering the data layer of a grid system, based on the grid definition and database theory, we can abstract several criteria of grid data management. First, any grid data access should be made transparently. *Transparence* refers to the data location and the form it is accessed. If multiple copies of the same data exist in the grid, *coherence* would be the second quality that the data management layer should provide, by guaranteeing that a certain well known coherence protocol, for all executions. As data must be available beyond application execution time frame, *persistence* ensures that data is safely and securely available on a contracted time frame. Last but not least, the data management layer should ensure a certain degree of *performance* while operating on distributed data sets. Next, we briefly highlight some of the existing solutions for grid shared data handling.

The *Globus middleware* defines a data management concept based the notion of *data catalogs*. Globus implements a set of protocols and utilities for data transfers and services for meta-data management. Globus toolkit provides a reference implementation for the standardized transfer protocol adapted for data transfers on the grid called GridFTP [79]. GridFTP is based on the FTP protocol and optimized for wide scale transfers, security and multiple parallel data streams. It supports stripped and parallel data channels, partial files, independent and automatic TCP buffering and progress monitoring. On top of GridFTP protocol, several other data management services have been built. These include Reliable Transfer Protocol (RFT) and Distributed Replication Service (DRS). RFT address the problem of reliable file transfer based on check-pointing information in a database and recovery mechanisms. Basically a reliable transfer is started and automatically carried on by the system. In case of a failure the transfer is automatically restarted and carried on to its completion. The transfer status could be checked either by subscribing to a notification or by polling. Replicated file management is provided by the Replica Location Service (RLS) which is simply a registry where logical associations are made between logical file names and physical files. Services like DRS make use of this information to replicate files among different locations. Considering the criteria of the grid data management layer, one can notice that Globus fulfills only the persistence and performance criteria. Other important characteristics such as transparence and coherence must be explicitly programmed at the application level. More, it seems that the only support that Globus provides is simply file oriented leaving a big gap to in-memory data sharing applications. ***As a consequence, we conclude that Globus does not provide support to program and build efficient large scale data sharing applications.*** Most grid applications that are running on Globus rely on

external message passing concepts. Thus, in the grid landscape, there are very few solutions for large scale data sharing models.

The LOTS system [80] is a distributed shared memory solution designed for large object spaces implemented in C++. Its primary goal was to be able to accommodate a large collection of objects and supply a huge distributed storage. The core feature is the ability to dynamically map objects from the virtual memory to local disks, being the first distributed shared memory system to supply a storage space independent of the total memory process space, but limited to the local disk storage. LOTS uses a mixture of write-invalidate and write-update protocols to handle barrier and data synchronization. Although it uses a relaxed consistency model, namely scope consistency, LOTS seems to be only suitable for cluster computing and fast interconnected machines. As all tests were conducted in cluster environments, there is no evidence of its suitability for large scale grid computing.

Another solution for grid shared data programming that came to our knowledge is SMG [81], which stands for Shared Memory for Grids. Although the name is pretty self explaining, it appears that SMG exists only as a prototypical idea. More, as SMG implementation is designed to be based on message passing libraries, more specifically on MPICH distribution, it is clear that it bears the characteristics and limitations of MPI and thus is unlikely to be suitable for large scale grid systems. Even its authors made a note that the MPI implementation makes implementation hard due to its missing support for multithreading and all existing MPI multithreaded implementations are not grid enabled. The choice for the consistency protocol is entry consistency. Important information on SMG such as detailed design, replication policies, mutual exclusion handling and performance measurements are currently missing, but from its existing description, we could conclude that it does not represent a viable solution for grid shared data programming.

Teamster-G [82] is one of the few recent software distributed shared memory systems designed for the grid. Although is intended for the grid, Teamster-G is implemented directly on top of the Linux operating system and is integrated into the Globus system. At run-time, it constructs a virtual dedicated cluster based on the data sharing requirements and does not rely on resource reservations like in the Globus approach. Same as in case of the previous solutions, there is little information on the concrete design and performance measurements. Nevertheless, it is documented that Teamster-G relies on discovery mechanisms provided by Globus such as indexing services and implements its own shared data layer based on a proprietary operating system. There is no information available on shared data replication handling as well as algorithms for mutual exclusion handling across grids.

One of the very latest and most complete solutions for grid shared data programming is JUXMEM [83]. JUXMEM supplies a non-structured, low level shared address space based on peer-to-peer concepts and uses JXTA [54] peer-to-peer middleware implementation. As no shared data structures are provided, the user is in charge of mapping the flat memory storage to its data types and is in charge of this explicit conversion whenever data is accessed. Same as the other approaches, it relies on a relaxed memory consistency model. One of the drawbacks of this approach is the fixed replication scheme that bounds data replicas at creation or fault time and which does not consider the system dynamics such as data usage patterns. A second drawback that is also mentioned by its authors refers to the peer-to-peer middleware solution, JXTA which turned out not to be very efficient for grid systems. Although its authors have tailored some of JXTA's protocols, there is much room for performance improvements. Last but not least, even if a large set of tests have been conducted on French grids, most of the tests were executed over fast interconnections. There is very little evidence on JUXMEM's behavior in large scale

environments. We believe that due to the direct peer-to-peer middleware dependencies, it is unlikely that such a system would exhibit a very good performance over large latency connections.

Last but not least, Dedisys [84] is an interesting research project in the distributed systems domain that aims to construct dependable distributed computing systems. Closely related to distributed shared data programming problem, the project aims to improve availability of both service and data centric components (data sharing and services). It uses replication as a basic mean to achieve availability for the data centric part and resource redundancy for the service centric part. Dedisys takes a very different approach to most other similar projects and makes use of replication protocols in order to allow non-critical operations in system degraded situations when network partitions are formed. In those cases non-conflicting operations can be executed in all partitions even if replicas might diverge and integrity constrains are violated. Dedisys proposes a reconciliation protocol based on user supplied policies that are used in order to re-establish replica and constraint consistency after nodes rejoin and network partitions are unified. The system is basically working in three distinct modes: the normal mode where no faults occur, degraded mode when network partitions occur and reconciliation mode where partitions are unified. System-wide consistency can only be re-established if all nodes are reachable. If not all the nodes are reachable after reconciliation, constraint consistency is re-established within the new partition or constraint consistency is discarded and only replica consistency is re-established. Dedisys goes more in the direction of constraint consistency than on data consistency. The consistency constraints as well as the reconciliation part need to be specified by the programmer and implemented programmatically and are checked by the system on method execution granularity. Thus, the system is focusing more on the system state rather than on distributed data state. It is not clear to what extent the constraint consistency specification increases the application code complexity. Last but not least, it appears that the major focus is on availability, fault tolerance at the system level rather than on the performance of the distributed shared data mechanisms. Similar to previously mentioned projects, there is no clear analysis on the system behavior in the context of wide scale, large latency systems.

We have pointed out that there are few shared memory systems designed for the grid. Many of these systems were tested in particular environments that represent ideal scenarios of fast connected machines most of the times being grouped as high performance clusters. These testing scenarios are pretty far from the ultimate evolution point of tomorrow's grid. Thus, we aim to investigate the problem of distributed shared memory for grid systems and aim to provide a system specification that addresses the following main points we found missing in most of the existing solutions:

- ***Large scale system over large latency connections –*** we address a true grid that is large scaled and where large latency connections are dominant between machines located at large distances
- ***Relaxed consistency and type coherence –*** we address relaxed consistency models together with type consistency as we expect that relaxed consistency does not carry sufficient information on data usage
- ***Object oriented architecture –*** we address an object oriented architecture as the most promising interface towards the grid application programmer
- ***Quantifiable system validation*** – addresses both prototype based evaluation, as well as theoretical or through formal methods as a proof of concept for the system model

2 Distributed Shared Memory

Distributed environments cannot be constructed unless their components are able to communicate to each other. One approach for communication handling is message passing, which involves sending and receiving explicit messages from one entity to another. Message format has to be agreed by both sender and receiver and each entity has to know an identifier or the “address” of its counterpart. Another approach was introduced by the concept of ***distributed shared memory*** (DSM), which abstracts data sharing between loosely coupled processes running on computers that do not share physical memory. The processes see the DSM as a single address space, which in reality is built by smaller distributed private memories belonging to each component of the system. Thus, DSM exists only virtually.

One of the main issues in designing and implementing DSM systems is to ensure good performance for their applications. Nowadays, distributed systems are typically widely distributed and have a large number of machines. Any access to a piece of data abstracted by a DSM might trigger a large communication overhead due to data movement and synchronization over large latency connections. One widespread approach to compensate large latencies is caching, by replicating data across the networked entities and keeping their values consistent to a well known protocol. Thus, the local memory of some or all nodes has copies of the data shared by the DSM. When data is supposed to be retrieved by a process, it is first looked-up in the local cache and, if not found or not valid anymore, the corresponding data is fetched from one of the remote cached copies.

One of the first implementation of DSM was the Apollo domain file system [85] which allowed processes to map files into their address space. Since then, a lot of research has been done in this domain leading to several approaches in designing and implementing DSM systems. More than twenty years ago, new directions have been suggested in distributed computing [86]. These were directed to increase the number of deployed, real-life DSM systems, free and open source DSM, highly integrated DSM environments, better tools and better performance. Many of the topics highlighted two decades ago are still being on the research table today. This proves that advances in distributed computing do not fully suit users and applications needs and further improvements are necessary.

2.1 Design and Implementation Issues

Distributed shared memory implementations have been mainly done in software. There are only a few hardware solutions that have been developed in the context of shared multiprocessor architectures based on Non Uniform Memory Architectures (NUMA) architecture like the Dash [87] and PLUS systems [88]. Generally speaking, NUMA architectures are those that exhibit different access costs to the distributed data. Dash and PLUS systems consisted of processors grouped in several boards, each of them seeing a single address space containing the memory of all the other boards. The strategy of the first software DSM systems such as Ivy [89], Munin [90], Clouds [91] and Mirage [92] was to map the distributed shared memory into a special address range of their virtual address space. The same address range was used in all the machines which led to a uniform system. These systems are called paged-based DSM and they were providing services at the operating system level. First system that pioneered the era of the middleware based DSM was Orca [93], followed by Linda [66], JavaSpaces [67] and

TSpaces [68]. They introduced a middleware layer on top of the operating system, totally hardware and network independent that provided data sharing means among loosely coupled processes.

Both paged-based and middleware DSM systems can run programs designed for shared memories on architectures that do not have such kind of physical memory. In the first case the kernel of the operating system is responsible to trigger the proper page faults and retrieve the faulting data. This architecture does not make any assumptions on how the data is organized into the memory space. However, the middleware solution makes clear assumptions on the structure of the data being handled into the distributed shared memory. Thus, it enforces to build higher level abstractions as building blocks of the contained data.

In a DSM system, data is shared by registering it to the virtual address space. As potentially many machines are accessing the same data, it is more efficient to have copies of the data located in the neighborhood of the machines that are making the request. This approach is trying to hide potentially high network latencies for data access. Having several copies of the same data distributed into the system leads to the problem of locating data efficiently, maintaining the correct system view on the data and managing consistency between the copies. This implies that a synchronization protocol must exist to keep the whole system consistent according to its specification.

Designing and implementing DSM systems is a complex and challenging process. These systems raise a series of issues such as data location and replication between machines, data update protocols (coherence), data granularity and the structure of the shared memory space, replication strategies of shared data, trashing, heterogeneity and the consistency model provided to their clients. The above mentioned aspects are discussed in the following sections.

2.1.1 Structure and Sharing Granularity

A DSM system can be thought of a large scale replicated collection of data abstracted globally, that distributed processes are creating, updating and consuming constantly. The structure of a DSM system can be divided into the following categories, depending on how their internal data is structured [2].

Byte oriented DSM systems are accessed as regular memory via unstructured read and write primitives that are operating on a memory location. Such systems are data representation independent and typically supported by the kernel of the distributed operating system. One of such systems is Ivy [89].

Structured and object-oriented DSM systems provide a data or object oriented view and operate at the structured data or object level. Their content can be changed only by manipulating these abstractions. Structured data was introduced in the Munin [90] system and object oriented approach was introduced by Orca [93] and Clouds [91]. The structure of the DSM can be defined and managed either at the distributed operating system level or at the middleware level.

Immutable-data DSM systems provide a collection of read-only objects that can be changed only by pulling them out of the DSM domain and pushing the updated data back. This approach is typically implemented by a middleware layer of the DSM such as Linda [66], JavaSpaces [67], TSpaces [68]. Locating the data in the space is done by associative lookup means following the tupple concept introduced by Linda. Interestingly, programming immutable-data abstractions require a programming mind shift, which according to its initiators could support any computational model [69].

Independent on the data structure, one common problem is defining the size of the shared data unit. This becomes more important in the context of wide data replication where any operation can potentially transport a significant amount of data and thus lead to severe performance penalties. In case of paged based systems, the exchange unit is the page size. Research has shown for page based systems that keeping page size smaller than a regular 4Kb page size does not necessarily improve the performance [89]. It is clear that a large data update would benefit from a larger page size, but updating a single bit in one page requires sending the entire content of that page to be updated at the other machines. In practice it is common that a shared page size is determined based on the physical pages defined in the operating system. Systems like Munin [90] allow updating parts of their pages and avoiding copying the entire page. Having larger data sharing units or pages increases the chances of false sharing and trashing. These issues are discussed in the next section.

2.1.2 Coherence Protocols

As data is replicated and multiple copies of the same data exist in a DSM system, there is the need for a synchronization and notification protocol to keep the data consistent between machines. Two major approaches have been imposed on propagating updates: *write-update* and *write-invalidate*.

The *write-update* protocol consists of sending multicast update messages to other replicas which are supposed to update their data. This protocol ensures that each replica data holder has the most recent data available and read operations can precede locally without any delay. The algorithm relays on the ordering of the update events. Depending on the delivery order of the multicast messages, different behaviors can be expected from the system. One of the problems is that the multicast protocols are expensive to implement in software [2] and reliable systems use hardware solutions like the Amoeba multicast protocol [94]. In case of page based DSMs this protocol is practical to be implemented only if the writes can be buffered. An improvement of the write-update protocol, although the idea is the same, was applied in the Munin [90] system and consisted in passing updates incrementally so that not the entire page was updated, but only the part that had been changed. Such a case favors multiple writer protocols where each writer is operating on independent pieces of data.

The *write-invalidate* protocol implies that when data is written by a process, all the copies are sent an invalidate message. Any process trying to read data that has been invalidated, has to retrieve the latest data from the DSM. Data that is not invalid can be read without any delay. Due to this fact, read-only data can be very easy replicated across the system. One problem of the write-invalidate protocol is due to *trashing*. This happens if one process is repeatedly writing data while another is reading the same data content. This producer-consumer pattern leads to frequent page copying from one process to another. In such a case the write-update protocol would be more efficient. Such a situation can be found in the Munin [90] systems, where data is proactively sent to the consumer when it is altered by the client, before the consumer requests it. It can be considered that an application might take advantage of one protocol for some data and the other protocol for some other data. Choosing the coherence protocols has been enforced at both system and application level. Trashing could be avoided at the application level, by properly organizing shared data and minimize copying penalties. Some systems like Munin [90] provide special constructs that allow passing additional information to the system in order to improve performance. Trashing could be also caused by *false sharing*. *False sharing* occurs when two or more shared variables are located in the same shared unit and are accessed by different processors. Although

there are not related, the whole page has to be synchronized to both processes as if they are accessing the same variable.

Write-invalidate has been implemented by designating an owner for each shared unit such as a page and maintaining a copy set of machines that have copies of the replicated data. When a process attempts to read or write a page for which it has no access permissions, the page is copied from its owner. If the requestor is a writer it becomes the owner of the page and the copy set is invalidated. The copy set is replaced by the new owner. This algorithm is known as the *shrewd algorithm* and was presented by Kessler and Livny in 1989 [95].

The above algorithm does not handle the problem of locating the owner of a page and maintaining the copy set for each page. Several algorithms have been devised to handle the copy set for a page. First approach is to use a centralized manager that stores all the copy sets and owners of all pages. Each process has to query the central manager in order to get the owner and copy set. The second approach, that alleviates the bottleneck of the central design, involves having multiple managers allocated for a given virtual address range. Based on the location that is supposed to be accessed, one of the managers is used as in the previous situation. A more advanced approached is the dynamic distributed manager. In this case the owner of a page is located by following a linked list of hints that are constructed as probable owners of the page. It is ensured that for n machines there are at most n-1 read operations from the hint chain to retrieve the correct owner [89].

2.1.3 Replication Algorithms

Replication is the process of maintaining the same entity content at different locations, providing thus a higher availability on data requests. Transferring and copying data from one machine to another happens according to a protocol that is well known by all machines. Basically, a sharing unit (e.g. page) can be moved from one machine to another or could be replicated, leading to more copies in the system. Depending on these options, four possibilities appear [96]. Migration and replication is implemented either using operation transfer or state transfer procedures [97]. The transfer operations could be classified as synchronous and asynchronous for distributed systems which correspond to eager and lazy in the database community [98].

Non-replicated, non-migrating blocks strategy represents the simplest and less efficient approach. Any system that keeps the blocks locally is sequentially consistent because at any time there is a single copy of the data in the entire system. All accesses have to be serialized to the page owner which creates a bottleneck and leads to no parallelism potential. Locating such block is very simple as a direct mapping function could be statically created.

Non-replicated, migrating blocks strategy implies that there is a single copy of the block in the entire system which can eventually move from one location to another. Having one single copy in the system, sequential consistency is achieved in this case too. Application might take advantage of this model to exploit a better locality for the shared data blocks. Still, the method does not support any degree of parallelism and it suffers from the trashing problem as data blocks move from one node to another. Locating the owner of the page can be implemented by either broadcasting or making use of a centralized or distributed server algorithm. As broadcasting does not scale well especially for wide area distributed systems and the centralized algorithm represents an obvious bottleneck, the only viable approach for locating the owner is the *dynamic distributed-server algorithm*.

Replicated, migrating blocks approach relaxes the replication constraint and alleviates the problem of lacking parallelism by allowing multiple copies of the data at the same time. This is a common approach for most DSM systems. Based on this general approach several layers of replication domains could be constructed, depending of the architecture of the system. Data changes are handled via the write-update or write-invalidate protocols. Locating the owner of a page can be implemented by broadcasting, fixed or dynamic distributed-server algorithm.

Replicated, non-migrating blocks refers to a system which allows multiple copies of the data, but their location never changes. This model can provide sequential consistencies if, for example all write operations are sequenced by a global sequencer.

All DSM systems that have replicated and migrated data have the problem to replace eventual pages if the local memory usage reaches an upper limit. This is a common problem in all caching systems. DSM systems apply basically two techniques: usage based and space based [96]. Usage based approaches measure the frequency pages are used. They apply replacement algorithms such as LRU (least recently used), FIFO (First-In-First-Out) or randomly. Space based replacement involve replacing data based on its location and not its usage pattern. The later approach is not found too often in DSM systems.

2.1.4 Replication Decisions

Some of the fundamental questions related to replication decisions are when, what and where to replicate. Some systems are taking a fixed static decision, but others (mostly middleware based solutions) take dynamic decisions based on system monitoring. The decision to replicate or not to replicate can be seen as a system assessment, where the following factors are considered: read/write ratio and statistics, communication path latency, response time, bandwidth and shared data/object size.

In general, replication tries to take advantage of temporal, geographical or spatial locality metrics based on the data sharing pattern. Such locality patterns determine when and where to replicate in order to improve a set of locality factors. As previously said, replica placement and access pattern identification can be evaluated through system observation by monitoring for example the data access response time and bandwidth consumption. Besides the replication strategies highlighted in 2.1.3, the location to replicate is a factor which can influence the performance of the overall system. For example, a data item could be replicated at each endpoint where it is required. Another decision is to replicate it only to the client which has the highest request rate, remaining that other clients request the data from that client. Other approach would be to replicate the data among the path from the client to the data owner, creating thus a chain of replicas. Depending on the type of the running applications, these replication decisions could play a decisive role in the overall behavior.

2.2 Consistency Models

The *consistency model* defines the system's responses to read and write operations on distributed shared data. In other words, the consistency model represents a contract between the running software and the memory which guarantees that if the software complies with certain rules, the DSM behaves as stated in the specification. One example described in [2] illustrates a system that maintains information about the CPU load in a distributed environment. As such information is updated quite frequently and becomes inaccurate very fast, there is no point to keep an up to date value of the CPU

loads. Typically users expect a strong consistency behavior similar to a local system, where each read returns the latest value written to that location. An isolated system that violates that expectation is immediately suspected as malfunctioning from the memory correctness point of view. However, distributed applications are operating on distributed data and do not have an accurate global clock, meaning that strict global ordering is not possible. Due to the high latency and low bandwidth, update operations from several machines to another might arrive in different order that they have been issued in respect to the absolute clock.

The consistency model has to deliver a clear answer regarding the value read by a process from the shared memory space. Basically, it aims to identify the write operation which has written the value returned by a given read operation. In other words, it specifies what shall be the returned value of a read operation, assuming that a series of write operations have been previously performed on different processes.

A hypothetical distributed system which provides absolute time synchronization could exhibit the strongest consistency model called *atomic consistency* or *strict consistency* [2]. Such system is said to be *strict consistent* if the value of any read operation returns the value written by the most recent write. As in real life distributed system there is no general agreement of what the most recent value is, such model requires the existence of an absolute clock which does not physically exist in distributed systems yet. Due to this strict requirement, *atomic consistency* cannot be provided in practice. To define more precisely the *atomic consistency*, we make use of the notion of *linearizability* by the following definition:

Definition 2.1: *A replicated shared object is linearizable (and having atomic consistency) if for any execution the following conditions are satisfied:*

1. *The interleaved sequence of operations meets the specification of a single correct copy of objects.*
2. *The order of operations in the interleaving is consistent with the real times at which the operations occur in the actual execution.*

In practical terms, the definition above translates as following: any read to a memory location x, R(x), returns the value stored by the most recent write operation to x, W(x).

2.2.1 Generic Consistency Models

The strongest consistency model used in distributed systems is the ***sequential consistency*** which was introduced by Lamport in 1979 [99] and relaxes the ideal atomic consistency model. In such a system all processes see the same order of all memory accesses. Lamport's definition can be rephrased as following:

Definition 2.2: *A system is sequentially consistent if for any execution the following conditions are met:*

1. *The interleaved sequence of operations is such that if Read(x)a occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is Write(x)a, or no write operation occurs before it and a is the initial value of x.*
2. *The order of operations in the interleaving is consistent with the program order in which each individual client executed them.*

Sequential consistency can be easily implemented if a global operation ordering could be ensured. Having a single entry point to handle all read and write requests would provide a global ordering

mechanism, but such solution is quite inefficient and exposes the single point of failure. It is enough to ensure that no memory operation is started until all previous memory operations have completed.

The ***causal consistency*** [100] relaxes the previous model more by specifying that all processes see only those memory operations in the same order that are causally related. Any other types of accesses can be seen in any order. They define the causally related variables as the variables that influence their values in any way. A typical example is computing a variable by the value of another one. Implementing such a model requires constructing and dynamically maintaining a memory dependency graph. Based on [100], we adopt the following definition for a causal consistent system:

Definition 2.3: *A system is causally consistent, if writes that are potentially related must be seen by all processors in the same order. Concurrent writes may be seen in a different order on different machines.*

Lipton and Sandberg proposed the ***pipelined random-access memory consistency (PRAM)*** [101] in 1988. This model relaxes the previous constraints by allowing the write operations of one process to be seen by the others in the same order as they are issued. All the write operations issued by the other processes are seen in different orders. One of the biggest implications is that processes do not agree anymore on the operation ordering as in sequential consistency. Lipton's model can be defined as following:

Definition 2.4: *A system is PRAM consistent, if writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

A further extension was proposed by Goodman in 1989 [102] as the ***processor consistency*** model. Processor consistency basically states that processes see the same write operation ordering for the same location they are operating on.

2.2.2 Synchronized Consistency Models

Synchronized consistency models are based on the observation that distributed applications don't need to see all the changes done by other processes, but only in some isolated points they require a consistent view of a part of the distributed memory space. This observation implies that better performance can be achieved if memory accesses are organized in groups. These classes of weaker consistency models relax the constraints further and obey the semantics of stronger consistency models such as sequential consistency in some particular conditions. These models make use of synchronization operations which give information to the system whether to propagate or not data and to allow or forbid other processes to access the protected data. Sections outside of the synchronization area are left inconsistent, assuming that the application does not rely on the accuracy of the data in those execution contexts. Dubois et al. proposed in 1988 the ***weak consistency*** model [103]. This model introduces the concept of *synchronization variables*. These variables are used to mark code blocks of memory synchronization. We can define a weakly consistent system as following:

Definition 2.5: *A system is said be weak consistent if the following conditions are met:*

1. *All accesses to synchronization variables are sequential consistent.*
2. *All previous write operations must be completed everywhere before an access to a synchronization variable is allowed.*
3. *All previous accesses to synchronization variables must be completed before access to a non synchronized variable is allowed.*

All the previous models imply that the entire DSM content is synchronized at the stage where a synchronization point occurs. At that point, all individual changes made by one processing node are sent to other nodes and all changes made by all other nodes are sent to the node which is about to synchronize. The idea to separate the above two operations led Gharachorloo to propose the ***release consistency*** model in 1990 [104]. The release consistency makes clear distinction between entering and leaving a critical section, by defining *acquire* and *release* synchronization variables. Release consistency might also be realized using the barrier concept. A barrier is a synchronization point that blocks all process execution until all the processes arrive at the same execution point. Acquire could be considered as the barrier arrival and release as the barrier exit point. Going back to the idea of release consistency, propagating the changes made by one process to other nodes need to occur when the process leaves a critical section. Changes made by other processes have to be propagated at the time the critical section is entered. Based on Gharachorloo's proposal, the following definition applies:

Definition 2.6: *A system is said to be release consistent if the following conditions are satisfied:*

1. *All accesses to acquire and release synchronization variables obey processor consistency semantics.*
2. *All previous acquires performed by a process must be completed successfully before the process is allowed to perform a data access operation on the memory.*
3. *All previous data access operations performed by a process must be completed successfully before a release access done by the process is allowed.*

Gharachorloo proved that properly labeled release consistency programs could achieve sequential consistency. Thus, it is the application's responsibility to properly organize its memory accesses depending on the consistency level to be achieved. The release consistency model has been further improved by Keleher in 1992, leading to the ***lazy release consistency*** model [105]. The novelty of this model is that it delays updating the changed variables from a critical section until some process is requesting the data. In release consistency, during the release operation all the modified variables are sent to all other processes, whereas in lazy consistency, these are sent on demand, when a process acquires access to synchronization variables. It has been proved that this approach generates less traffic on the communication system and offers better performance.

A more relaxed consistency model, proposed in 1993 by Bershad, is the ***entry consistency*** model [106]. This model links every shared variable to a synchronization object, thus when a process acquires a lock it is guaranteed to get the latest value of the variables bound to it. Write handling is done by defining a write lock, allowing that only that particular process could write the variable during the critical section. The model is defined as following:

Definition 2.7: *A system is said to exhibit entry consistency if the following conditions are met:*

1. *An acquire operation of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
2. *Before an exclusive mode access to a synchronized variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.*

All the above relaxed approaches put some burden on the programmer to define the correct synchronization variables and critical sections. The ***scope consistency*** model [107] is an attempt to automate the association between shared variables and synchronization objects. The core idea is to use the variable local scope in order to make automatic associations between shared variables and synchronization objects. Although the association is implicit, the programmer has to manage the proper scope by organizing variables in nested constructs.

Several other researchers have followed this direction, addressing particular aspects of these models, but no major improvements have been made. One of the latest novelties in consistency models is ***view consistency*** [108] which defines disjoint views on the DSM space. Views are accessed using acquire and release calls, pretty similar to the entry consistency model. Similar to the entry consistency model, there is a data-synchronization variable association, but in case of the view consistency, it happens when the view is created. Although the authors of view consistency claim that the model focuses on the data and operates on the data view without locking the data explicitly, one cannot think only in terms of views without taking into account which kind of data the view consists of. Thus, a certain connection between the abstract view and the data is always present in the programmer's mind. The implication of this observation is that view consistency exposes another interface of the entry consistency model with a main difference in shared data creation and protection (acquire/release operations).

2.3 A Programmer's View

Similar to message passing constructs, DSM provides another kind of abstraction layer that exposes other interfaces and concepts to programmers. At the lowest architectural layer, a DSM system has to send and receive messages in order to communicate to other processes. One major issue in message passing is that message passing can happen only if all the communication peers are active at the same time. The message passing model involves marshaling and un-marshaling data at each process end-point. Besides marshaling, message passing requires that data is packed and unpacked in messages at each end-point. On the other side, in DSM systems, shared variables do not need to be marshaled, except for different architectures. Thus, DSM systems tend to be more homogeneous than their counterpart. DSM programming seems to shield more the programmer from the underlying messaging architecture, providing thus the ability to handle higher level abstractions [96].

Different abstractions have different impacts on programmers. While message passing makes programmers aware of the potential cost of a message passing call, DSM calls are most of the time transparent. Depending on the implementation, proper additional constructs have to be used in the source code to indicate special handling of DSM data. If this small but effective information detail is overlooked, significant performance degradation can easily occur. For example, using weaker consistency models requires programmers to annotate their source code with special constructs that define the consistency protocols for those variables. For example, in the Munin system, a programmer can use seven types of variables: read-only, migratory, write-shared, producer-consumer, result, reduction and conventional. Depending on these constructs, different synchronization algorithms are applied allowing better performance achievements. This means that the consistency model affects programmability as it is used by programmers to reason about their programmed interactions.

Efficiency and security are two characteristics of distributed systems that are enforced in most systems. Message passing offers isolated communication between processes as their address spaces are distinct. On the other side DSM provides a shared environment where data partitioning and access

regulation have to be enforced additionally, otherwise any process can read any piece of shared information. In case of message passing systems, security can be handled only distributed, to each entity taking part of the communication flow. On the other side, due to DSM system's shared environment, security can be handled in a single place, namely the data access component, which is independent of the number of shared data clients.

Efficiency has been addressed for a long time in DSM. Experiments in ***small scale systems*** have proved that in many cases these systems can perform close to their message passing equivalents on the same hardware [109], [110]. However, there are plenty of applications which exhibit poor performance when a DSM implementation is being used. Many factors are influencing their performance, most of them being due to the shared data structure, data interaction pattern and application deployment. The consistency model plays also a great role, but no general valid comments can be made in this direction. One application might have very good performance using one consistency model and poor results using another one. Depending on the application type, a consistency model which offers good parallelism should provide good overall performance as well. One major challenge of this work is to provide good performance for certain distributed applications deployed in large scale shared data systems, by making use of relaxed consistency models.

3 A Model for Distributed Shared Objects on the Grid

3.1 Considerations

Grid applications are wide area distributed applications that are using the grid infrastructure as an execution environment during their life cycle. Some of the grid applications are simply running on the grid without any knowledge of the running environment. Such applications are simple or classical applications, mostly batch or job based that are taking advantage of distributed and parallel environments. Most of these fall in the category of legacy or resource hungry applications. Other applications are more evolved and are making use of the grid infrastructure and services, thus they are grid-aware applications. Until recently, grid applications have been monolithic, consisting mostly of compute intensive and batch aggregated code. Developing grid-aware applications is not an easy task as grid computing differs from conventional distributed computing in several aspects. First, there is a different scale factor that is continuously changing due to dynamically joining and leaving machines in different clusters. Second, grids have a different organization and structure which leads to different administration policies. Third, grid applications have different focus, being more performance and throughput oriented than simply running a certain code. Last, but not least, we can witness innovative applications every year in this domain, that are basically multi-disciplinary applications which aggregate resources from new members of virtual organizations.

In contrast to traditional distributed and parallel computing where computing resources are easily identified and most of the times have predefined locations (e.g. clusters), grid systems exhibit by definition another scale dimension. One cannot imagine a fixed grid, but rather a distributed system where machines are dynamically changing their membership and new machines are joining the infrastructure continuously. As an example, research grid systems have been initially located at some university centers where different departments have been joined together to create one of the first grids. As a next step, universities from different cities have joined the research grid and later on the grid has spanned across a continent or even across continents. Such a large scale distribution cannot be witnessed in any cluster or traditional distributed system. This kind of wide area distribution leads naturally to different administration and security policies. Most of the grid members have their own policies that must be preserved when putting their resources for the common use. Fortunately, grid systems address the problem of security as a core built-in feature, but it still has to be applied which implies an inherent managing and runtime verification overhead.

Grid systems expose several constraints and special conditions. For better understanding, one can think of a grid like a multi-level hierarchical structure that can be modeled as a non directed graph. Each node represents a machine or a group of machines. Typically, a group of machines is a cluster or LAN where each machine can communicate with another one within the same group with the same known and upper bound communication latency. Thus we consider together all machine groups and depict them as a single group node as shown in Figure 10. Such hierarchical structure is constantly getting deeper (more levels) and wider (more groups) during the evolution of a grid system. A unique characteristic is the unpredictable layering as a result of unpredictable joining and leaving groups, plus changes in physical communication channels. For example, a certain group is connected over the air using wireless LAN or via Bluetooth to different hosts, thus it might appear in different layers at different times. Besides the unpredictable layering, another characteristic of the model is the unknown bandwidth

and latency of each communication line between two arbitrary groups. To simplify the discussion, we omit the failure model and consider that there is an upper bound latency and a lower bound bandwidth.

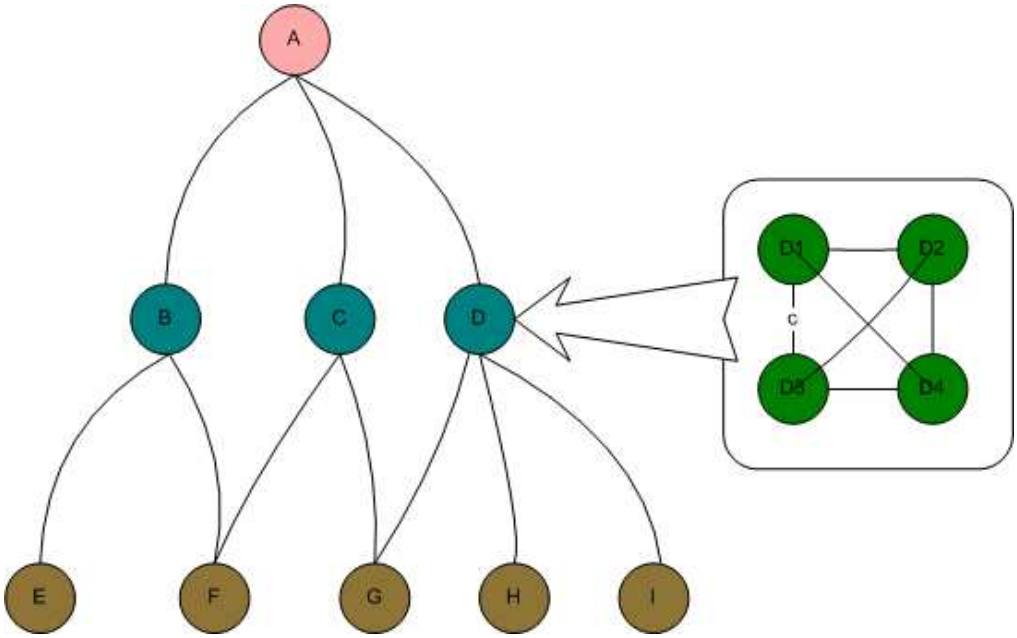

Figure 10**:** Grid Layering

As interactions happen between components from different layers and groups, which are typically connected by large latency communication channels, we consider that highly synchronous operations are not desirable in these circumstances due to performance issues. Such operations might take place locally, inside a group like *{D1, D2, D3, D4}* for example, but should be limited or simply restricted between any components that communicate over a large latency path. If for example, some distributed state must be consistently maintained between different high latency layers, such guarantees would be difficult to maintain using synchronous communication. In such a structure, typically the identity of *D1…Dn* nodes is known between one another, but not to external nodes like G, H or I. A natural consequence of high latencies is bigger communication delays. Two approaches that we are following in order to overcome the problem of communication delay from different perspective consists in reconstructing the problem and partition into acceptable communication groups with known latencies and providing useful programming information so that the run-time system can take advantage of semantic information (e.g. meta-information) and apply dynamic optimizations. Both ideas are not new, but to our knowledge they have not been applied together on a grid system before. More, the second idea has not been applied in the context of grid programming and grid scale distributed shared data. Network partitioning is an approach followed by MPI implementations and hierarchical distributed algorithms to optimize communication according to the network topology information and providing meta-information is a well known approach in many software engineering domains like formal verification and testing.

Having seen that grid systems exhibit a series of constraints and conditions that are hard to cope with, one can ask what the reasons are for programming grid applications. The reasons stem from the nature of the application. Applications might need to exploit their distributed and parallel nature, by decomposing into smaller entities that are distributed on the grid. Other applications would simply want to decrease the processing time, using the grid as an execution vehicle for their huge set of operations. Some other reason is the fact that an application cannot be run on the architecture it resides, thus it

makes sense to execute it remotely on a grid, that provides the environment (e.g. parallel architecture) and necessary components and services to run the application successfully. Last but not least, the ultimate reason is to use the grid service oriented architecture at its best, by exploring the connection between application components and required grid resources. However, to support an easy grid application process, one needs a grid programming model that simplifies application development. Such a model should contain at least a high level expressive communication abstraction, flexible data sharing abstractions that can be efficiently implemented on the wide scale grid architecture. There is still debate if the support should be part of the programming language or at the run-time system.

3.2 System Model

Most of the currently available grid programming models rely on traditional or foreign programming models that are adopted via a technology porting process. Here we include MPI like libraries like MPICH-G2, file based distributed data services based on GridFTP and higher level data access services like OGSA-DAI. Services and high level abstractions for programming shared data structures on the grid are almost not present on the grid programming model landscape. ***Thus, our idea is to propose a grid service layer for shared data programming which provides a distributed shared memory system and its corresponding programming model adapted for the grid***. Some of the core reasons for considering such a model stem from the drawbacks of message passing solutions that put an additional burden on the programmer to decompose the computation handle load balancing and explicit communication orchestration and the lack of automatic data layout and optimization support.

Although many distributed shared systems have been developed in the last two decades, most of them are limited on a certain number of nodes and work best in a fast interconnection network. Such systems do not qualify for the grid as they do not fulfill the scalability and wide range deployment requirements. Blindly applying such model on the grid will most probably fail to provide the expected behavior and reasonable performance. Distributed shared data items must be widely shared and the problem of managing the consistency of mutable data on wide area systems is raised.

Some of the previous attempts in designing DSM for the grid have used logical mappings over one single large machine group. One concrete example is the JUXMEM [83] approach, where peer-to-peer groups have been spawned across the grid. Even the authors of JUXMEM recognized that the wide distribution of peers in the overlay layer is problematic and current overlay implementations such as JXTA [54] have serious performance issues in largely distributed environments. Thus, we argue that another split is necessary, which clearly identifies the connection points into the entire grid universe. We see this mapping as part of the system deployment, instead of relying on a predefined mapping.

As suggested in previous sections, in order to address thousands of nodes, we decompose the system into a federation of clusters called universes. The logical representation of a universe is homogeneous and communication latency in a universe is typically small and bound to a higher known margin. The physical entities that form a universe could be heterogeneous (e.g. machines with different resources and operating systems). Communication outside the universe, which occurs between universes, is unknown, but still it has an upper limit. Following, we give a definition of the universe and its building blocks.

Definition 3.1: *A Node is a physical stand alone execution environment on a networked machine which can both execute an arbitrary number of programs and which can communicate to other known nodes through an agreed communication protocol.*

Definition 3.2: *A Universe Node is a logical environment that belongs to a physical node that is able to accommodate a number of arbitrary data items where following conditions apply:*

1. *Every universe node n has a fixed data storage capacity denoted as C(n).*
2. *All accommodated data items are available until the node is alive.*

Definition 3.3: *A Universe is an execution and data storage environment consisting of a collection of interconnected nodes and a subset of their universe nodes, where the following rules apply:*

1. *For any two nodes within a universe, there is a direct communication link between them.*
2. *Every communication link within a universe has a known and constant latency and bandwidth which are constant during a certain time interval t.*

Definition 3.4: *A Grid Universe is an execution and data storage environment consisting of a collection of several Universes where the following rules apply:*

1. *For any two universes, there is at least one connecting communication path between them.*
2. *Every direct connecting link between any two universes has known upper and lower bound latency and bandwidth which are constant in a given time interval t.*

Definition 3.5: *A Grid Application is a collection of one or more processes and data where the following rules apply:*

1. *Processes are distributed across a set of nodes*
2. *Each universe node is owned exclusively by the application.*

A *Universe* is a logical collection of machine nodes which provides a hosting environment for distributed objects. Nodes are homogeneous and have a data storage capacity in memory and code execution capabilities. Each node can hold a certain number of objects so that the sum of all object weights held by the node shall not exceed the node's capacity. All existing universes form together the *Grid Universe*. Each *Universe* is a continuously evolving entity together with its connections to the other universes. A *Universe* groups together more physical machines which share the same communication paths, thus the intercommunication channel in a *Universe* is homogeneous and has known and constant characteristics within a prescribed time frame. Communication between universes is unpredictable, unknown and dynamic. As an example of a concrete universe, one can consider a physical cluster or a LAN and a grid universe as several interconnected clusters. The universe abstraction is a dedicated shared grid data environment that corresponds to one application. If we take for example two different applications, each would access its own grid universe without any interaction between the two universes at the logical level. Of course, at the physical level, there might be overlaps between them as some nodes from one universe might be mapped on same machines from the second universe.

3.3 Basic Programming Model

Following previous considerations, we propose a basic programming model, based on the concept of distributed shared data, that serves as a starting point for analyzing and defining the complete

grid shared data model. The model we propose is object oriented which provides flexible interfaces for data encapsulation. An object oriented model is a natural and convenient way to abstract data sharing objects and separate the interface from the implementation, supporting the idea of objects residing in architectural different run-time systems like nodes in universes. In addition, security policies can be easier enforced and integrated to an object than to a flat data structure. The model revolves around the concept of the *Universe*, which has been introduced in the previous section.

The Grid Universe acts as a container for grid objects and provides means to create, delete and locate grid objects based on a unique object identifier. The users do not operate directly on objects, but rather on object references. A grid object reference is a handle to a concrete grid object that provides the same interface as the object provides. Following, we define both the grid object and the grid object reference.

Definition 3.6: *A Grid Object is an object hosted by the Grid Universe with the following properties:*

1. *Each Grid Object has a globally unique and location independent identifier called GID.*
2. *Each Grid Object has an explicitly associated and unique identifier called OID.*
3. *Each Grid Object can be part of only one Universe at a time.*
4. *Each Grid Object o has a weight associated denoted as W(o), which refers to its memory space demand.*

The Grid Object has two identifiers associated to. One is the GID which is always associated by the system and cannot be given by the object user. The second is the OID which is always given by the object creator and is intended to serve as a human friendly identifier. The OID can be used later to lookup a certain grid object.

Definition 3.7: *A Grid Object Reference is a handle to a grid object, that provides the same interface as the grid object and which satisfies the following conditions: If Gr1 and Gr2 are references to the same object Go, then GID(Gr1)= GID(Gr2) = GID(Go) and OID(Gr1) = OID(Gr2) = OID(Go).*

The basic Grid Object that we introduced above serves as the basic building block for handling data objects on the grid. As the definition does not imply anything about the payload and object's functionality, we define specialized objects based on the grid object definition.

Definition 3.8: *A specialized grid object is a grid object that extends the interface of the grid object by providing additional methods and data.*

Basically, the specialized objects carry data and provide interfaces to execute certain operations on the object. The specialization can be seen as either through inheritance following the terms of object oriented concepts, when a specialized object extends the basic object through additional methods and members. An alternative is to define the state of the object as a generic type and let the object store external data. The disadvantage of this approach is that the object state type is fixed and coarse grained which provides a non-intrusive view on the object internals. Generally speaking, the objects we considered are passive objects. Their specialized methods are invoked within an execution context provided by the running environment, thus processes and grid objects are considered orthogonal.

As presented earlier in this chapter, in order to decrease access time to grid objects from different universes, a common technique used in distributed computing is data replication for performance. If some conditions are satisfied, a grid object would be replicated to other universes, provided that object state

can be transferred from one process to another across a communication path. Following, we introduce the notion of replicated grid objects:

Definition 3.9: *A Grid Object Replica Gr of a grid object Go is a grid object copy of Go, with the same interface and data, hosted by the Grid Universe with the following properties:*

1. *GID(Gr) != GID(Go)*
2. *OID(Gr) = OID(Go)*

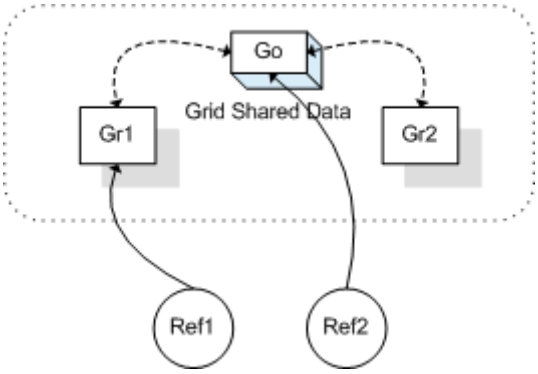

Figure 11**:** Grid Object Reference Handling

As a consequence of the grid object replica definition, in conjunction with the grid object reference, if one uses a grid object Go that has two replicas Gr1 and Gr2, then for any two references Ref1 and Ref2 pointing to either Go, Gr1 or Gr2, OID(Ref1) = OID(Ref2), but GID(Ref1) != GID(Ref2). In other words, a reference points always to the correct object, but the object could be the primordial object or any of its replicas, as replicas might exist within the same universe where the original object lives or any other universe. There is no guarantee that a reference points always to the primordial object, but it is assured that it points to the object with the given identifier (OID).

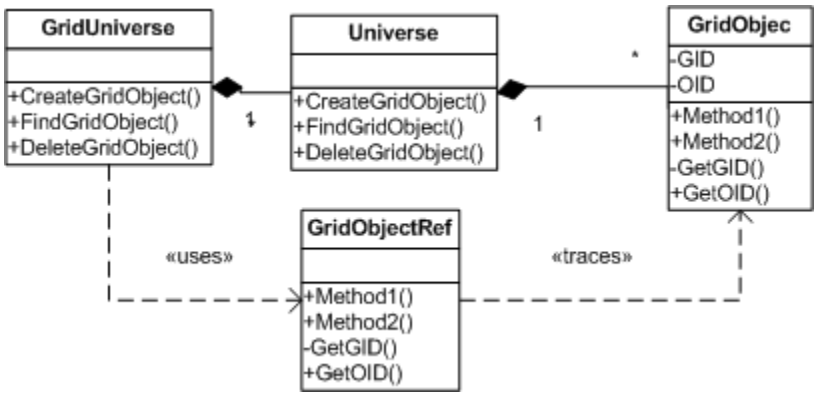

Figure 12**:** Grid Universe Abstractions

The logical relationship between the universe, objects and object references is depicted in Figure 12. The Grid Universe contains a collection of objects that are living in multiple universes. When an object is created, a reference is returned to the user. Thus, in order to invoke a grid object method, it is necessary that the calling process binds the desired object to an object reference. The reference can be used to locate any replica of the object designated by the identifier of the reference pointing object. One can consider the grid universe as a collection of distributed universes where objects are living. Some

object replicas might exist within the same universe or might exist in different universes. If there is no synchronization between object states, those replica objects can evolve in parallel. At synchronization points the state is synchronized between the object replicas.

The semantics of GridUniverse's methods is described next, considering that one defines a specialization of a concrete grid object and its reference as MyGridObject and MyGridObjectRef.

- ***GridObjectRef GridUniverse::CreateGridObject(GridObject obj)*** – creates a concrete grid object on the GridUniverse based on the given GridObject obj. In case of success it returns a GridObjectRef, otherwise null.

```
MyGridObject mgo = new MyGridObject()
mgo.Initialize()
...
MyGridObjectRef mgr = (MyGridObjectRef) GridUniverse.CreateGridObject(mgo)
```

- ***GridObjectRef GridUniverse::FindGridObject(OID oid)*** – returns a grid reference to a grid object represented by the provided identifier id. In case there is no grid object with the given identifier, or the operation failed, the method returns null. In case of replicated objects, the returned reference might point to any replicated object or the primordial object.

```
OID oid = GetTheId() // the method retrieves an object id (e.g. via message passing)
...
MyGridObjectRef mgr = (MyGridObjectRef) GridUniverse.FindGridObject(oid)
if (mgr != null){
   ...
}
```

- ***Boolean GridUniverse::DeleteGridObject(OID oid)*** – removes the given grid object from the universe, including all its replicas. Returns true on success and false if the given object does not exist or the operation failed.

```
OID oid = GetTheId() // the method retrieves an object id (e.g. via message passing)
...
Boolean ok = GridUniverse.DeleteGridObject(oid)
```

In the simple grid model presented above, clients operate on data entities that are always referred via the grid object reference. A grid object reference is a handle to the storage locations of the grid shared data. Such references can be passed between processes via standard communication mechanisms or embedded into a grid object as transport data.

3.4 Motivating Scenarios

Different grid applications have different data sharing requirements and patterns. In this section we present several grid application scenarios, where distributed shared data is used in different ways. Based on the use cases we aim to define a grid shared data programming model and the consistency model designed for the grid shared data service layer.

To illustrate some scenarios, we consider that the Grid Universe provides synchronization mechanism by making use of synchronization variables following the critical region concept, in the following manner:

- *Acquire(GridObjectRef)* – synchronizes all grid objects that are pointed by the ref grid object reference and enters a critical region where one can operate exclusively on a reference
- *Release(GridObjectRef)* – marks the end of the critical section and releases the reference from being exclusively owned by the current owner

3.4.1 Distributed Order Placement

Problem: Order systems consist of a collection of products and orders grouped in an order catalog. Products are continuously added by warehouses and delivered to the customers according to the orders they made. Each order contains several products. Customers create orders by selecting one or more products and adding them to the order. At a given point in time a product's state such as the price is updated by the company. This should be reflected in each submitted order that contain the given product, but shall not propagate to already sent orders. Similarly, a product could be removed from the market. This fact should be also reflected in the each pending order.

Challenge: The system must guarantee the correct consistency as specified in the system model.

Approach: Orders and products are modeled by Order and Product abstractions represented by objects. A catalog contains all the orders placed into the distributed system. An order object aggregates several products. When processing the order, product information is synchronized and it is verified that the product is valid by checking if the corresponding product reference is valid.

Pseudo-code:

```
Process1:
   OrderRef order = GridUniverse.CreateGridObject(new Order(“100”))
   GridUniverse.Acquire(order)
   ProductRef p1 = GridUniverse.FindGridObject(OID(“Product1”))
   order.addProduct(p1)
   ProductRef p2 = GridUniverse.FindGridObject(OID(“Product2”))
    order.addProduct(p2)
   ProductRef p3 = GridUniverse.FindGridObject(OID(“Product3”))
   order.addProduct(p3)
   GridUniverse.Release(order)

Process2:
   OrderRef order = GridUniverse. FindGridObject (OID(“100”)) // get order 100
   // No other process changes the order
   GridUniverse.Acquire(order)
    For each productRef in order
         GridUniverse.Acquire(productRef)
         //... Process product -> Updated product data values
```

```
      GridUniverse.Release(productRef)
  Endfor
  GridUniverse.Release(order)
```

3.4.2 Command and Control

Problem: Given a network of sensors, each sensor is able to execute specific commands and return results. The sensors are distributed on a wide scale in a grid. There is a corresponding process running for each sensor which knows the communication protocol with its assigned sensor. A sensor process reads through a global reference the commands and puts the result back to a location referenced by another global reference. Each sensor is able to process only one command at a given time. The global references for both commands and results are globally known. Clients want to get data out of the sensors. The client machine would write the necessary commands for each of the sensors interested in. The results are collected and assembled by the client process. The client must make sure all the processes are free and no processing is taking place (no command is present in the referenced command data).

Challenge: The system must provide means to lock on multiple resources and avoid deadlock situations.

Approach: We make the following considerations:

S1,…Sn – sensors 1..n

RefCi – a shared reference for the sensor i where the command has to be written

RefRi – a shared reference for the sensor i where the result of the command is written

Pseudo-code:

```
Client Process1 – interested in sensors 1 and 2
Wait until RefC1 is empty, RefC2 is empty
GridUniverse.Acquire(RefC1)
GridUniverse.Acquire(RefC2)
// write command for sensor 1
WriteCommand(RefC1)
// write command for sensor 2
WriteCommand(RefC2)
// release references
GridUniverse.Release(RefC2)
GridUniverse.Release(RefC1)

Client Process 2 – interested in sensors 1 and 3
Wait until RefC1 is empty, RefC3 is empty
GridUniverse.Acquire(RefC1)
GridUniverse.Acquire(RefC3)
// write command for sensor 1
WriteCommand(RefC1)
// write command for sensor 3
WriteCommand(RefC3)
```

```
// release references
GridUniverse.Release(RefC3)
GridUniverse.Release(RefC1)

Sensor process i
While (true)
   Wait until RefCi is not empty
   GridUniverse.Acquire(RefCi)
   // read command
   Command = ReadCommand(RefCi)
   // process command via the sensor communication protocol
   Result = ProcessCommand(Comand)
   // release sensor data
   GridUniverse.Release(RefCi)
   // Write result
   GridUniverse.Acquire(RefRi)
   RefRi.SetData(Result)
   GridUniverse.Release(RefRi)
End While
```

3.4.3 Environmental Data Repository

Problem: A data set (contained in a database for example) is distributed over the grid as distributed shared data. The data represents environmental information like temperature, wind conditions, humidity etc. Grid nodes perform operations on the grid data and supply results. Operations are normally triggered by grid clients. Requests are serialized into a queue. One operation might imply reading multiple data (e.g. to determine a specific condition one must take into account condition from location A and B). This means that environment conditions A might be required in parallel by different processing grid nodes. New information appears from time to time that needs to be updated. If no environmental data updates occur, the reading speed shall not be affected.

Challenge: If there are no changes in the environmental data, parallel processes must not unnecessarily be blocked.

Approach: We consider are two types of locks: reading locks, that allow parallel read accesses and writing locks. If a write lock is enabled all reading locks are blocked. Multiple read locks have no blocking effect without a write lock.

Pseudo-code:

```
Client Process:
   CommandRef = GridUniverse.CreateGridObject(command)
   QueueRef = GridUniverse.FindGridObject(“CommandQueue”)
   GridUniverse.Acquire(QueueRef)
    QueueRef.addCommand(CommandRef)
   GridUniverse.Release(QueueRef)
```

```
Processing process:
   // get first command from the queue
    QueueRef = GridUniverse.FindGridObject(OID("CommandQueue"))
   GridUniverse.Acquire(QueueRef)
    CommandRef = QueueRef.First
   CommandData = CommandRef.GetData
   GridUniverse.DeleteGridObject(CommandRef)
    GridUniverse.Release(QueueRef)

   // process command
   For each EnvDataRef in CommandData
      GridUniverse.AcquireRead(EnvDataRef) // read lock
      // Process data
      GridUniverse.Release(EnvDataRef)
   End For

Environment Data Sync process:
// request write access to the environmental data storage i
EnvDataRef = GridUniverse.FindGridObject("EnvironmentDataI")
GridUniverse.AcquireWrite(EnvDataRef) // write lock
// update data
GridUniverse.Release(EnvDataRef)
```

3.4.4 Parallel Genetic Algorithm

Problem: Optimization problems with multiple contradictory constraints for which there is no known deterministic algorithm are suitable for a genetic approach. Speedup could be achieved by applying a parallel version of the sequential genetic algorithm.

Challenge: Some processes are working on shared data (disjoint populations) that are not supposed to be used by other processes at the same time. During some phases references are accessed only from the same process. Later on, data can be exchanged between processes and the cycle continues.

Approach: Following the classical genetic algorithm approach, a given number of random possible solutions are generated. Solutions are grouped into populations. Each population evolves independently and from time to time solutions migrate from one population to another in a random fashion. After a given number of steps the best solution is retrieved.

The following notations apply:

Pj – Population j

Si(Pj) – Solution i in population j

RefS(Si, Pj) – a reference to solution i, in population j

N – the size of a population (constant)

M – total number of populations

Initially one process generates Si(Pj) where i=0, n-1 and j=0, m. Thus the total number of solutions is n*m which evolve on m populations hosted on m processing units.

Pseudo-code:

```
Worker Process:
// a worker processes a population
For(i=0, i < totalsteps)
   // apply evolution within the population
   For (int k=0; k < random(N / 20,  N/ 10)
        // get 2 random population members
        RefS1(Si, P) = GetRandomReference(CurrentPopulation)
        RefS2(Sj, P) = GetRandomReference(CurrentPopulation)
        // No locking/synchronization is necessary since solutions
        // are accessed by the same process
        // apply crossover
        Crossover(RefS1, RefS2)
        // apply mutation
        Mutation(RefS1, RefS2)
   EndFor
   // Must ensure that the other population is after crossover/mutation phase
   Wait for all processes  to reach this point
   // exchange x references with population k
   For(i=0,i<10)
      RefS1(Si, P) = GetRandomReference(CurrentPopulation)
      RefS2(Sj, Pk) = GetReference(RandomPopulation)

      // aquire both references
      GridUniverse.Acquire(RefS1)
      GridUniverse.Acquire(RefS2)
      // modify their population membership (exchange solutions)
      RefS1.SetPopulation(k)
      RefS2.SetPopulation(currentPopulation)
      GridUniverse.Release(RefS1)
      GridUniverse.Release(RefS2)
   EndFor
EndFor
```

3.4.5 Distributed Builder

Problem: Multiple independent processes are working in an assembly line fashion, by operating on one data object one at a time. The processes are either operating on the same part of the object or on disjoint parts.

Challenge: A grid shared object membership is continuously transferred between calling processes. Other objects can be manipulated in parallel by multiple processes. Processes must synchronize themselves at some points.

Approach: The client writes an object to be built into the grid universe. Builder processes are waiting for an object that they know how to build. In other words, each builder waits until there are objects in a given state. The object is retrieved and the corresponding phase is completed. Next, the object is taken by the next builder until the object is completely built.

Pseudo-code:

```
Master Process:
   // generate a product to be built
   // mark object as "migratory"
   Pr = GridUniverse.CreateGridObject(new Product(state1))
   WaitFor(signal) // wait for the end of build process
   GridUniverse.Acquire(Pr)
   //...get the data
   GridUniverse.Release(Pr)
End

Builder Process i:
   While(true)
      ProductRef = GridUniverse.FindGridObject(statei)
      // No need to synchronize if non-conflicting part is written
      BuildPart(ProductRef)
      If (LastBuilder)
         SetSignal(signal) // signal the end of computation
   EndWhile
End
```

3.5 Consistency Model

The memory consistency model represents a contract that the grid shared service has to satisfy at any time. It states what the value of a certain object is, among a set of wide replicated distributed objects, if certain conditions are satisfied. Choosing a specific consistency model has several impacts on the overall system. First, it regulates a certain degree of overlapped operations so that different processes are not blocked if they operate on the same data. At the same time the synchronization model is defined implicitly by the consistency model. Second, different consistency models imply different underlying operations which generate at the end different communication traffic patterns and volumes. Last but not least, consistency models have a visible impact at the programming level, meaning that different consistency models have to be expressed differently at the API level. Such a restriction limits the adaptability at the consistency level and as a result we have to adopt the most suitable consistency model for grid systems.

Based on the scenarios presented in the previous section, we consider as the most promising consistency model the *entry consistency model* [106], which is also the least restrictive model (or the most relaxed). In this model, synchronization happens between clearly defined operations: *acquire and release*. The drawback is that it requires additional programming effort to specify synchronization points. The rationale for this choice is that the entry consistency protocol assures data synchronization at entry

point in the synchronization code, avoiding thus the penalty of update protocols that generate a higher communication traffic pattern in a large scale environment.

Considering the grid universe model introduced in section 3.2, in order to illustrate the impact of the consistency model on the programming interface, we assume that a grid object is able to store some data which is accessed through some methods. In its ***raw form***, entry consistency requires that each shared object be associated with a synchronization variable such as a lock, which is defined as following:

Definition 3.10: *A lock is a synchronization object that defines the boundary of a critical region and has the following properties:*

1. *It is identified by a unique name.*
2. *Can be associated with one or more objects.*
3. *It defines the scope where the protected variables can be accessed by supplying an interface with the following operations: Acquire, AcquireExclusive and Release*

The following code sample shows how locks are used together with grid objects. As one can immediately notice, programming with entry consistency adds the requirement of an explicit binding between locks and the variables to be protected. This way, programming is more complicated and error prone as explicit locking and unlocking must be handled in pairs by the programmer. The advantage stems from the shared space separation and thus the possibility for a higher code execution parallelism, since multiple critical sections can operate on disjoint shared objects, allowing thus code to be executed simultaneously.

Pseudo-code:

```
Process1:
GridObjectRef g = GridUniverse.FindObject(OID(“13”))
Lock l(“Lock100”)  // Creates a lock
l.Bind(g)          // Associates the lock with the object referred by g
l.AcquireExclusive()
// here we can update the object’s state
l.Release()

Process2:
GridObjectRef g = GridUniverse.FindObject(OID(“13”))
Lock l(“Lock100”)
l.Bind(g)
l.Acquire()
// here we can read safely the object’s state
l.Release()
```

The entry consistency makes clear distinction between lock types, providing exclusive locking for state updates and regular locking for reading the object state. Semantics and operation sequences obey the entry consistency model which is defined as following in [111]:

Definition 3.11: *A system is said to be entry consistent if the following conditions are met:*

1. *An acquire access of a lock is not allowed to perform with respect to a process, until all updates to the guarded shared data have been performed with respect to that process*
2. *Before an exclusive mode access to a lock by a process is allowed to perform with respect to that process, no other process may hold the lock, not even in non-exclusive mode*
3. *After an exclusive mode access to a lock has been performed, any other process' next non-exclusive mode access to that lock may not be performed until it has performed in respect to that lock's owner.*

The above definition can be reformulated more informally as following. When a process invokes a lock's acquire operation, the call may not execute immediately until all the protected objects by that lock have been brought up to date. Second, before modifying a shared object, a process must enter a critical section in exclusive mode to assure that no other process might update the same data. Third, if a process tries to enter a critical section in non-exclusive mode, it has to check with the owner of the lock to retrieve the most recent copies of the shared objects.

Considering the fact that the protected grid data are objects and processes have to use the same locking variable in order to synchronize a certain object, a simplified model can be devised based on the previously introduced model as described in Figure 13, leading to clearer interaction as given in the next code snippet. A minor extension has been made by introducing the timeout value for both acquire operations. The semantics is that if the object cannot be acquired within the prescribed timeout, the acquire operation fails and the object is not acquired. If no timeout value is specified, a default value is taken.

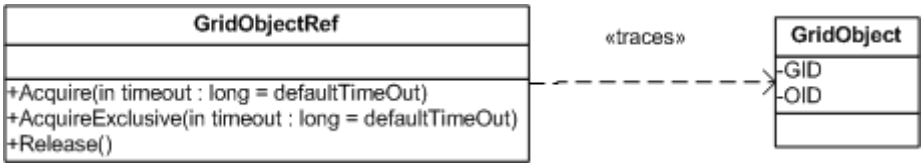

Figure 13**:** Grid Object Synchronization

The effect of this proposal is of an implicit one-to-one mapping between a synchronization object and the corresponding lock, leading to an object locking model. The semantics of acquire and release operations remain unchanged as introduced above, with the remark that the lock is the same entity as the grid object reference. This model is similar to the view-based consistency model [108] where the data associated to a view is one object. In the original proposal of the view-based consistency, the view data is unstructured. However, using the object oriented model, one can associate several objects to a view by applying object composition. Although some similarities exist to the view-based consistency model, we stick to the entry-consistency and object oriented definition of our model.

Pseudo-code:

```
Process1:
OID oid("13")
GridObjectRef g = GridUniverse.FindObject(oid)
g.AcquireExclusive()
// here we can update the object's state
g.Release()
```

```
Process2:
OID oid(“13”)
GridObjectRef g = GridUniverse.FindObject(oid)
g.Acquire()
// here we can read safely the object’s state
g.Release()
```

The consistency model specifies only the correct response of the system at the client inquiry. The system can be optimized by trying to maximize parallel operations. The consistency model regulates the object state synchronization, but it does not specify how the synchronization is achieved. Additionally, there might be grid objects that have a particular usage pattern that might allow us to apply further optimizations. For example, it could be that one object is accessed only by a process for a determined period. As no other processes are accessing the same object at a certain stage, one can imagine only one object in the entire universe where the object state need not propagate to the entire system. Generally speaking, the system needs to be told which operations can be performed in parallel. For example a reader-writer interaction can be relaxed in case the write protocol is write-shared and individual parts of the shared object can be modified concurrently. Another optimization beyond the scope of the consistency model is by exploiting a synchronization pattern such as “in advance” or “delayed” synchronization. This means that the grid universe simply anticipates that some object will be required by some processes in the future and could perform thus data copying in advance. At the other pole, some data might not be needed so often, thus their synchronization can be delayed until they are required. Last but not least, the information flow for the consistency protocol can be tuned, so that operations are grouped as much as possible leading to bulk communications rather than isolated and sporadic interactions.

3.6 Specialized Objects

As presented in section 3.2, we tackle the problem of grid shared data in two distinct dimensions. First, we have introduced the consistency model in section 3.5 as the base for object state synchronization and correctness. Second, we follow the object usage pattern, in the idea of communication and object replication optimizations. Here we address type specific coherence based on the observation that different classes of objects are accessed in different ways and the access pattern might be changing during the process lifetime. Building blocks of this model start by understanding the grid data sharing use cases and synchronization patterns. Based on the use cases presented in section 3.4 we aim to abstract mechanisms for type based coherence, based on user provided information. Together with the consistency model, the type based coherence model we devise in this section aims to be the base for the grid data sharing concept of this research.

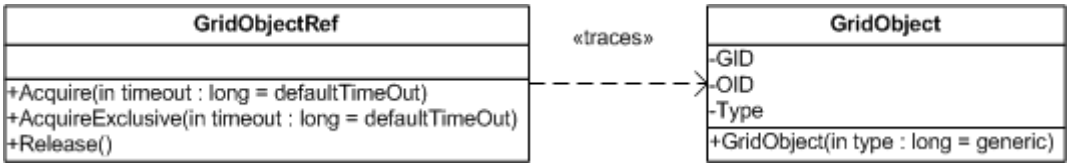

Figure 14**:** Grid Object Interface

Every parallel programming model has to define abstractions and concepts for object synchronization, which besides the consistency model implies mutual exclusion concepts and conditional

synchronization. The programming model we aim to define introduces different types of grid shared objects and synchronization mechanism in order to give the run-time system useful information to allow a higher concurrency degree and to minimize wide area communication overhead. We consider next the following grid object types and their semantics, following the specialization chain. To illustrate the creation of different object types, we consider that each GridObject is bound to a type at creation time, as illustrated in Figure 14.

3.6.1 Read-Only Objects

Read-Only objects are immutable grid objects that are created by one process and their value is bound to the value at the time of creation. These kinds of object do not require any synchronization mechanisms as the state is not changing after the object is added to the grid universe. Such kind of objects can benefit of a high replication rate.

Definition 3.12: *A read-only object is a grid object accessible via its corresponding reference with the following properties:*

1. *The value of the object is bound to the creation time value.*
2. *Object value changes are not propagated among object replicas.*

Interface semantics: *A read-only object's Acquire, AcquireExclusive and Release operations have do-nothing semantics as they don't require any synchronization.*

Rationale: The object specification carries only behavioral semantics and does not specify any concrete synchronization interface. The system can exploit through replication the fact that read-only objects are not supposed to be updated. Thus, any read operation from other universe than the one where the object lives may trigger the object to replicate into the calling universe. This kind of specification is very useful for algorithms which operate on large input data that is consulted very often. Read-only objects can be realized based on regular grid objects introduced in section 3.3. This kind of objects can be used for example in the matrix multiplication case.

Pseudo-code:

```
Process1:
MyObject x = new MyObject(READONLY, OID("13"), 100) // initial value is 100
GridObjectRef p = GridUniverse.CreateObject(x);
a = p.GetValue()     // a = 100
x.SetValue(200)
a = x.GetValue()     // a = 200
b = p.GetValue()     // b = 100

Process2: (Runs after Process 1 created the object "13")
GridObjectRef g = GridUniverse.FindObject(OID("13"))
b = g.GetValue()     // no need for acquire; b = 100
g.SetValue(150)      // Programming error! Local variable shall be used instead
b = g.GetValue()     // b = 150
GridObjectRef g1 = GridUniverse.FindObject(OID("13"))
b1 = g1.GetValue()   // 150 or 100. Depends on which replica g1 points to
```

3.6.2 Private Objects

Private objects are objects that belong to a certain fixed location, namely a fixed node. During a computation, sometimes a grid object is only needed by the node where it belongs to and no other process from the same universe or other universes require that object. Their purpose is to apply local optimizations for frequent local interactions or to mark that the object is fixed. As private objects are accessed typically only locally, they do not have grid scope locking mechanisms as they are not replicated and are typically accessed by only one process. Such an object can be thought of as simply a local data carrier. The following definition applies:

Definition 3.13: *A private object is a grid object accessible via its corresponding reference with the following properties:*

1. *It is bound to a fixed node.*
2. *Only one copy of the object exists at any time in the Grid Universe.*

Interface semantics: *A private object's Acquire, AcquireExclusive and Release have the same semantics as the ones of the generic GridObject.*

Rationale: Although they are shared objects, they are typically accessed by one or more local processes. Some algorithms can benefit out of this sharing pattern if they exhibit a waveform processing pattern. Such condition happens if they are processing elements of a structure and after the element is processed, it is only used by the local processes and no other external process. Thus, the reason of this object type is to reduce the synchronization overhead by reducing the scope of object monitoring and provide an optimized local lock mechanism.

Discussion: The implications of the above definition are that a grid object can be transformed into a private object which behaves as a unique object with a central synchronization mechanism within the object itself. If the object had replicas at the time of the transformation, the replicas are removed from the grid universe. Normally, an application does not create such object type, but it changes the type of an existing grid object. Private objects can be based on regular grid objects introduced in section 3.3.

Pseudo-code:

```
Process1:
MyObject x = new MyObject(PRIVATE, OID(“13”))
GridObjectRef p = GridUniverse.CreateObject(x)
p.AcquireExclusive() // local scope locking
p.SetValue(1oo)
p.Release()
…
Process2 (deployed on the same node as Process1):
GridObjectRef g = GridUniverse.FindObject(OID(“13”))
g.Acquire()       // local scope locking
y = g.GetValue()
g.Release()
g.AcquireExclusive()
g.SetValue(15)    // local scope locking
g.Release()
```

3.6.3 Migratory Objects

Migratory objects represent grid objects that are accessed in phases by multiple processes. In every phase, a single process is taking exclusive ownership of the object. After the object is used by one process, another process takes its turn and applies another state modification. Migratory objects carry only the semantics of the object type and take advantage of the exclusive acquire operation to trigger a migration of the object to a new location for the new access.

Definition 3.14: *A migratory object is a grid object accessible via its corresponding reference with the following properties:*

1. *The object is accessed by multiple processes in phases, one process at a time.*
2. *For any migratory object, there is no replicated object in the grid universe.*

Interface semantics: *A migratory object's Acquire, AcquireExclusive and Release operations have the same semantics as the ones of the generic GridObject, with the following difference: AcquireExclusive triggers the object to migrate to the universe or universe node where the calling process resides.*

Rationale: The object specification carries behavioral semantics and shares the same interface with the basic grid object. Migratory objects are not replicated, but are migrated whenever a different process than the one that holds the object is calling the AcquireExclusive method. Migratory objects can be used in case of the parallel genetic algorithm.

Discussion: The consequences of the migratory object definition relate to the replication protocol and object usage. These kinds of object are never replicated and only one process is using them at any time, thus no concurrency issues can be exploited, but only locality Similar to private objects, they are supposed to be used only by one process at a time. The difference is that whereas private objects are fixed (they don't change their location), migratory objects can move between nodes or between universes. In terms of object usage, any process that is interested in reading such an object's data is simply using the normal *Acquire* operation for a read lock. Different to private objects, all locks are performed remotely, at the object scope, at the location where the object is located.

Pseudo-code:

```
Process1:
MyObject o = new MyObject(MIGRATORY, OID("13"))
GridObjectRef p = GridUniverse.CreateObject(o)
p.AcquireExclusive() // signal that the object is to be used on this node
p.SetValue(10)
p.Release()          // release the object

Process2:
GridObjectRef p = GridUniverse. FindObject(OID("13"))
p.Acquire()       // the object is simply read, no migration happens
x = p.GetValue(); // x = 10
p.Release()

Process3:
GridObjectRef p = GridUniverse. FindObject(OID("13"))
```

```
p.AcquireExclusive()      // the object is requested to migrate where Process3 resides
p.SetValue(20);
p.Release()
```

3.6.4 Producer-Consumer Objects

Producer-Consumer objects are grid shared objects written by only one process called producer process and read by multiple other processes called consumer processes.

Definition 3.15: *A producer-consumer object is a grid object accessible via its corresponding reference with the following properties:*

1. *The object's state is written by only one process.*
2. *The object's state is read by multiple other processes.*

Interface semantics: *A producer-consumer object's Acquire, AcquireExclusive and Release operations have the same semantics as the ones of the generic GridObject.*

Rationale: The object specification carries behavioral semantics and shares the same interface with the basic grid object. The runtime system can take advantage of their semantics and can perform eager object synchronization. This implies that after a write operation that releases the object, the object's state might be synchronized in advance so that all other reader processes do not require another internal state synchronization. Thus, the possible optimizations in this case would be object replication for read operations and eager updates to all replicas at release time. In the best case scenario, consumers do not need to wait until the state is replicated across universes. Producer-consumer objects can be based on regular grid objects introduced in section 3.3.

Pseudo-code:

```
Process1:
MyObject o = new MyObject(PRODUCERCONSUMER, OID(“13”))
GridObjectRef p = GridUniverse.CreateObject(o)
x = 10
While (true){
   p.AcquireExclusive() // AcquireExclusive signals the producer process
   p.SetValue(x)
   p.Release()
   x = ComputeSomething()
}

Process2:
GridObjectRef p = GridUniverse. FindObject(OID(“13”))
While (true){
   p.Acquire()          // consumer
   x = p.GetValue()
   p.Release()
}
```

```
Process3:
GridObjectRef p = GridUniverse. FindObject(OID(“13”))
while (true){
   p.Acquire()       // consumer
   x = p.GetValue()
   p.Release()
}
```

3.6.5 Read-Mostly Objects

Read-mostly objects are those grid objects that are mostly read than written, leading to a high read/write ratio. In case of these objects, it is desirable to have more replicas that can be updated after each write operation requested by a process. This kind of objects is similar to producer-consumer objects, with the difference that there might be more than one writer as in the producer-consumer case.

Definition 3.16: *A read-mostly object is a grid object accessible via its corresponding reference with the following properties:*

1. *The object’s state is written by at least one process.*
2. *The object’s state is read by multiple processes.*
3. *The object’s read/write ratio is higher than a given threshold, thus the object is mostly read.*

Interface semantics: *A read-mostly object’s Acquire, AcquireExclusive and Release operations have the same semantics as the ones of the generic GridObject.*

Rationale: The object specification carries behavioral semantics and shares the same interface with the basic grid object. The runtime system can take advantage of their semantics and can perform a proactive and eager replication protocol to achieve shorter synchronization timings. Such objects can be used in the use case of the keep-alive communication.

3.6.6 Result Objects

Result objects are objects that are constructed through a builder process, where many processes are writing separate and non-conflicting parts and one process is reading the final result upon completion. Once written, they are only used by *one* process that collects the result.

Definition 3.17: *A result object is a grid object accessible via its corresponding reference with the property that object’s state can be decomposed into distinct, non-conflicting parts.*

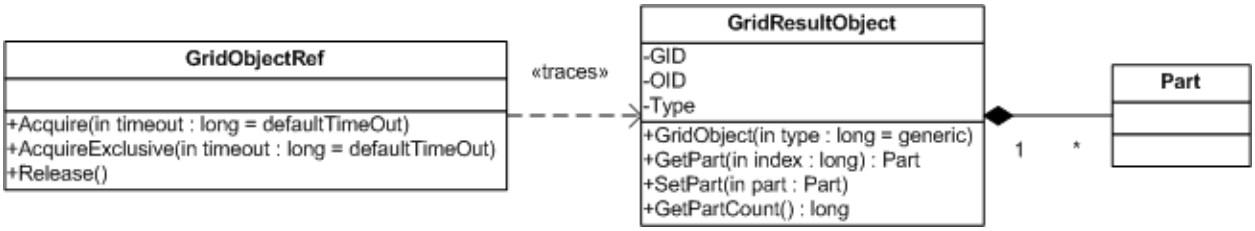

Figure 15: Grid Result Object Interface

Interface semantics: *A result object's Acquire, AcquireExclusive and Release operations have the same syntax as the ones of the generic GridObject, with the following semantic differences:*

1. *AcquireExclusive shall be used when processes are writing non-conflicting parts of the object*
2. *Acquire performs object state composition (object state is synchronized when Acquire is issued)*
3. *Additional methods are provided in order to manipulate its parts: GetPart to return the object part from a given index, SetPart to set an object's part to a given index and GetPartCount to return the number of the object's parts.*

Rationale: The object specification carries both behavioral semantics and specific interface description. The benefit is that such objects can relax the synchronization constraints when object's state is updated, if the object can be decomposed in disjoint parts. Writing any of these parts does not require any specific synchronization and can run in parallel. When the object state is collected by the "reader" process, the state is synchronized by following a global merge procedure. The Acquire operation can be also used to indicate the state when inconsistencies are not tolerated by the running process. Result objects can be applied to the distributed builder use case.

Discussion: The implications of the above definition are no locking mechanism is used when processes are writing non-conflicting parts of the result object. Whenever object parts are written, *AcquireExclusive* shall used to signal that a part of the object has been modified. Whenever object state is needed, *Acquire* shall be issued so that the state of the object distributed among the universes shall be assembled.

Pseudo-code:

```
Process1:
MyObject o = new MyObject(RESULT, OID("13"))
GridObjectRef p = GridUniverse.CreateObject(o)
p.AcquireExclusive()
Part p1 = p.GetPart(1)  // retrieve one part
p1.SetData(…)           // update the part
p.Release()

Process2:
GridObjectRef p = GridUniverse. FindObject(OID("13"))
p.AcquireExclusive()
Part p2 = p.GetPart(2)  // retrieve one part
P2.SetData(…)           // update the part
p.Release()

Process3:
GridObjectRef p = GridUniverse. FindObject(OID("13"))
p.AcquireExclusive()
Part p3 = p.GetPart(3)  // retrieve one part
P3.SetData(…)           // update the part
p.Release()
```

```
Process4:
GridObjectRef p = GridUniverse.FindObject(OID(“13”))
p.Acquire()                  // object state is put together
for i=1 to 3
   Part part = p.GetPart(i)  // retrive the correct part value
end for
p.Release()
```

3.6.7 Write-Mostly Objects

Write-mostly objects are grid objects that are frequently modified between synchronization points. As the read/write ratio is low, such objects do not benefit from a higher replication rate, thus a lazy synchronization protocol would probably work best. This kind of objects are similar to result objects, with the difference that the object state cannot be decomposed in independent parts, thus the entire object state must be kept synchronized between synchronization points.

Definition 3.18: *A write-mostly object is a grid object accessible via its corresponding reference with the following properties:*

1. *The object’s state is read by at least one process*
2. *The object’s state is written by multiple processes*
3. *The object’s read/write ration is lower than a given threshold, thus the object is mostly written*

Interface semantics: *A write-mostly object’s Acquire, AcquireExclusive and Release operations have the same semantics as the ones of the generic GridObject.*

Rationale: The object specification carries behavioral semantics and shares the same interface with the basic grid object. The runtime system can take advantage of their semantics and can perform a lazy synchronization protocol and bulk updates in order to avoid frequent communication due to the high write/read ratio. At the same time, replication factor of this kind of object shall be kept low.

3.6.8 Generic Objects

Generic objects are those that do not fall in any of the above categories. They follow the generic interface introduced in section 3.3 without any semantic information, thus no additional optimization is applied to this kind of objects. The consistency model of these objects falls in the system consistency model which is entry consistency.

3.7 Cost Model

As introduced in section 3.2, we consider the grid universe as a multilevel collection of universes which behave in a predefined determined manner. In a universe we can perform any operation in a small amount of time due to the fast communication paths between its nodes. However, inside a universe we cannot know the time distribution of node availability. As a result, special care has to be taken so that internal operations inside a universe do not suffer from potential node failures. Moving to the next level, in case of the connection between universes, we consider that the connection is dynamic and unreliable. In other words, one cannot assure that a certain connection holds for a predefined period of time or what

quality of service (e.g. bandwidth and latency) it offers. Figure 16 shows a more detailed view on the grid universe organization and the relationship between universes.

By assigning to each universe a node in a graph and drawing an edge between each node which share a physical connection we obtain the grid universe graph. The cost of each edge denotes the cost of the communication flow between two universes.

We consider that within a universe Ui there communication channel has a cost δi. The communication cost within Ui is given as a vector V, where for each i, V[i] = δi. Communication between two universes Ui and Uj has a cost φij. If there is no direct communication link in cases where the universes do not share a physical direct connection, the communication cost C(LGi, LGj) is given as the sum of all communication connection between universes which form a path from Ui to Uj. This model does not reflect the real life situation too accurate as a result of the possibility of different packet routing paths, but gives enough information to assess the communication costs.

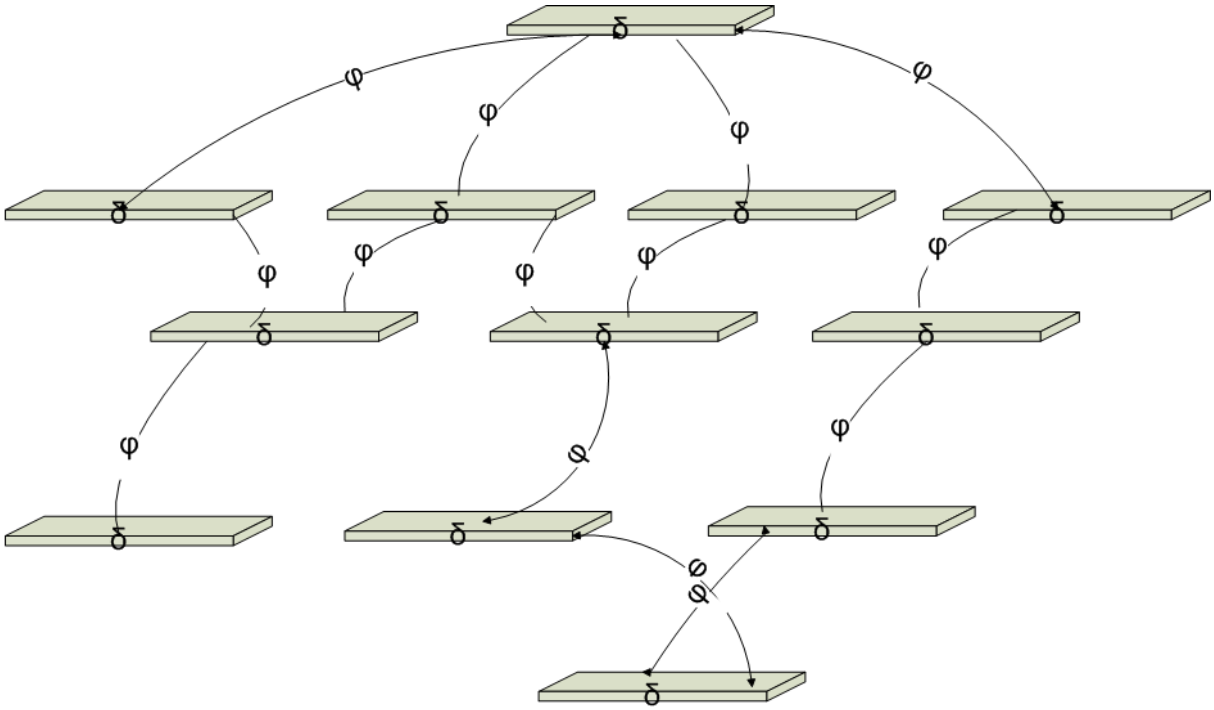

Figure 16**:** An Example of Possible Grid Universe Organization

We consider that at some universe Ui there is a process running, P on one node, which is operating on a grid shared data object O which happens to live at universe Uj. In case the object O does not migrate, we have a constant cost as the sum of all φmn, where (m,n) is an edge in the grid universe graph. In case object O is replicated to Ui, the cost decreases to Max(δi).

It is worth to note that the model is very dependent on the requests that the grid universe has to fulfill. Any grid object operation that occurs on a universe might cause a change in the subsequent access costs. On the other hand, evaluating the access costs from the point of view of a single running process is not relevant, as the grid is a collaborative environment where interactions are happening on wide remote areas. Thus, our model has to be refined in order to capture the state of the entire system, rather than of an individual process. In order to restrict the frame of the operation span across the grid, we define an

observation window Δ which denotes a time frame where the system is observed. We assume that within this window, all communication channels are fixed and not changing, thus all vectors V and the cost matrix is constant.

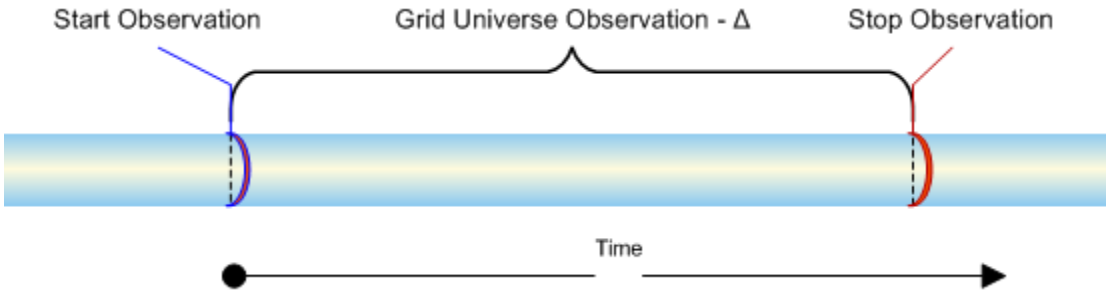

Figure 17: Grid Universe Observation

Under the mentioned conditions, we consider that n independent processes Pi (processes which have different codes) are spawned across the grid. Each process runs on its own grid context, potentially in distinct universes, but all are working together on a common goal. At each step each process is executing code and from time to time each process is accessing several grid shared objects, one at a time. Each grid shared object access has a cost ci, where i denote the access number. If we log all accesses by each process we obtain a sequence of access costs, for the period Δ as following:

$$Access(Pi) = \{ci \mid ci \text{ is access cost } i \text{ with } 0 < i < n \text{ and } time(n) < \Delta\}$$

We expect that such logging information can provide useful information for a better replication protocol and better locality of references in the grid universe. Also, such information can be used for offline system analysis. Logging has to be done at object level, capturing information like object id, process id, access type (read/write/synchronization), access time and cost. The most promising information we expect is to assess the programmer's decision towards the shared grid object types and to help improve their decisions towards certain object types. Such information might be used the change the type of the objects dynamically to improve the cost pattern and provide better runtime performance. We plan to approach this information as one of the next steps towards better grid shared data services.

4 Overall System Design

4.1 Assumptions

In order to address different system architecture solutions we make some assumptions to define the design boundaries of the proposed system and programming model realization. These considerations focus on the system separation, universe mapping to physical resources, universe structure, communication mechanisms, execution environment and replication policies.

Logical and physical universe mapping. A universe consists of a logical group of nodes that are mapped to physically interconnected machines that share for example the same physical connection such as a LAN. We always operate at the logical level and imply a clear distinction between the physical and logical nodes. During the deployment of the system, a mapping is required so that physical universes are defined across LANs or clusters. A good deployment tries to construct universes out of low latency interconnected machines residing in physical proximity of the same network. If this deployment guideline is not followed and large latency connections are present within a universe, the overall system performance could be negatively impacted. A sample of physical universe mapping is illustrated in Figure 18, where three networks are connected to each other and the forth is having just one connection point. Such a deployment can be seen at universities where several departments are interconnected via a VPN and an external entity is connected to one of the gateways through a secure connection.

Universe communication. The communication scheme can be divided into communication inside (between nodes) or outside (between universes) a universe. We assume that all nodes within a universe can communicate to each other in either point-to-point or multipoint mode. Communication between any two nodes that belong to two different universes (communication outside a universe) is done using point-to-point mode only. We believe that it is not efficient to use multicasting protocols across universes because of the large communication latencies. If, for example, multicasting can be supported efficiently in two distinct universes or they have similar latencies, the universes can be merged under one single universe. One real-life issue to consider is the case where universes are mapped to two physical networks that use network translation (NAT) which prohibits a low level point-to-point connection such as via TCP/IP. This limitation can be overcome by either using a proxy in every universe and route requests to each universe proxy for dispatching or providing higher level point-to-point communication interfaces that do not have such limitation such as a web service interface (higher level proxy pattern). Although this situation can occur and several solutions are known, it has been not considered explicitly.

Synchronous and asynchronous operations. We assume that nodes can communicate to one another through object oriented interfaces by invoking methods from one another via some of their public interfaces. Method invocation can be done in both synchronous and asynchronous way. For example, node N1 from universe U1 can invoke an asynchronous method on another node N2 from universe U2. The response shall be received by N1 within a timeout value and shall carry the data produced by the execution of the request. The execution semantics is “maybe once”, meaning that method invocation can fail. This execution semantics is specified by most of remote method invocation protocols such as RMI. We assume that group communication is also possible via object oriented interfaces and a node can invoke a method on objects from multiple nodes. The caller is able to wait for receiving all responses or just one of them (the first received reply). Thus, we assume that a node has the ability to perform simple and group communication with other nodes from the grid universe, through single or group method invocation.

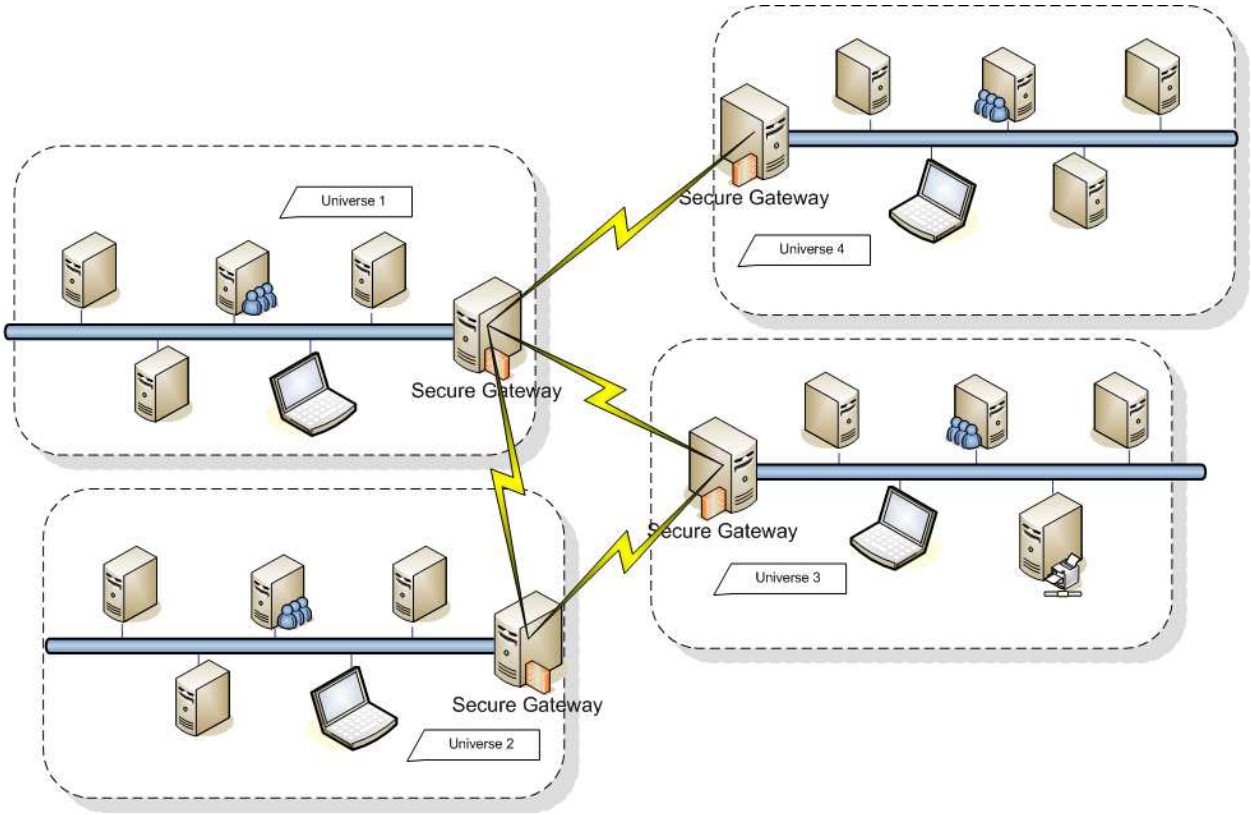

Figure 18**:** Physical Universe Mapping Sample

Universe discovery. By universe discovery we understand the operations carried out so that a universe knows of the existence of other universes. Within a universe, node discovery activities are required in order to dynamically join and remove nodes from a universe. The problem of universe and node discovery is described in each concrete solution.

Execution and volatility. We consider that the grid execution layer is responsible for spawning the application's code (possible several executables) across universes and their nodes. The grid universe provides the distributed environment to create and maintain shared data until the data is deleted explicitly by the application or the node dies. We don't aim to build a long term data persistence layer on the grid scope. The persistence level is in our case determined by the life time of each node that is holding certain grid shared data.

No faults. During a certain time interval it is always possible that some machines are turned off or simply experience some kind of faults that limits their proper functionality. Such a fault could be simply an exception in a communication module or a process crash. These kinds of faults can occur at any time in any universe, thus we can consider them as Byzantine events. In the design we do not consider explicitly such conditions, thus we don't take the fault tolerance aspect into consideration. For example if a grid shared object is created on one node and is never replicated to other nodes, in case of the node crashes, the data is lost. If the object has been replicated to another node and the replicated data is outdated (no synchronization has been performed yet) and the first node crashes, the system would contain the outdated copy of the shared data.

4.2 Replication Handling

We intend to use replication as the primary mechanism for performance improvement and not for fault tolerance. When data is created on the grid, we choose the closest node to the node who issued the "create shared data" request and which has enough capacity left to accommodate the grid shared object. The same node who issued the command can be chosen if it has enough capacity available. Upon data request during application execution, the grid shared data might be replicated to other nodes from the same or different universes in order to reduce access time due to large latencies. The system decides at run-time to replicate the data based on its specified replication policies. The replication policy follows a system definition of a rule based specification. This means that the same replication rules are considered for all the applications running on the grid. As the rule based replication policy definition is provided from the outside of the system (e.g. deployment information), it can be fine tuned differently to individual systems.

We define a function ***ShallReplicate*** that evaluates if a grid shared data object shall be replicated from node to another or from one universe to another universe. The function iterates through all configured replication rules and returns true if replication shall take place or false if the indicated object shall not be replicated. We consider some system predefined replication rules and these can be extended as required by the application deployment policies. We define next a small set of replication rules based on object types and system status information.

```
/** Evaluates system defined replication rules and indicates if replication shall be performed or
not.
@param  sourceUID      Universe replication source, where data to be replicated is located.
@param  sourceNID      Node replication source, where data to be replicated is located.
@param  targetUID      Universe replication destination, where data shall be replicated to.
@param  targetNID      Node replication destination, where data shall be replicated to.
@param  ref            Reference to the grid object that shall be replicated.
@return                Returns true if replication shall take place, or false if no replication
shall be performed.
*/
ShallReplicate(Universe sourceUID, Node sourceNID, Universe targetUID, Node targetNID,
GridObjectRef ref)
        for each rule in RuleTable
                if (Evaluate(rule, sourceUID, sourceNID, targetUID, targetNID, ref) = true)
                        return true
        end for
        return false
```

Replication primitives. As we introduced previously the idea of replication rules, we introduce some basic primitives that are used in the definition of the replication rules. We don't aim to present the complete possibilities of system built-in rules, but rather to illustrate how replication is performed in the system. The concrete replication rules will be presented when a concrete model is simulated and analyzed. As replication decisions are very important, we expect that depending on the chosen replication rules, different behavior shall be observed in the overall system as well as different performance figures.

```
/** Returns the type of the object referred by the grid object reference
        @param  ref    Reference to the grid object that shall be identified.
        @return        Returns the object type as following: READONLY, PRIVATE,
                       MIGRATORY, PC, READMOSTLY, RESULT, WRITEMOSTLY, GENERIC
*/
Type(GridObjectRef ref)
```

```
/** Returns the size granularity of the object referred by the grid object reference
       @param  ref    Reference to the grid object that shall be weighted.
       @return        Returns the object type as following: SMALL, MEDIUM, LARGE.
*/
Size(GridObjectRef ref)

/** Returns the total object count from a given universe.
       @param  u      Grid universe.
       @return        Returns the total object count.
*/
Count(Universe u)

/** Returns the total object count from a given universe with a given object type.
       @param  u      Grid universe.
       @param  oid    Object identifier
       @return        Returns the total object count.
*/
Count(Universe u, OID oid)
```

One observation about replication rules is that for any object identifier *oid* and any node *n*, ***Count(oid, n) <= 1,*** meaning that on any node, there is at most one instance of any object. As a result, a grid shared object cannot be replicated to a node where one of its replicas already exists. We consider other status information that might be used during the replication rule definitions, some of these are being suggested as following:

```
/** Returns the total acquire hits issued from a universe for a given object from another
universe.
       @param  source Grid universe where the grid shared object resides.
       @param  oid    Object identifier.
       @param  caller Grid universe from where the acquire invocations have been issued.
       @return        Returns the total acquire hits
*/
AcquireHits(Universe source, OID oid, Universe caller)

/** Returns the total acquire exclusive hits issued from a universe for a given object from
another universe.
       @param  source Grid universe where the grid shared object resides.
       @param  oid    Object identifier.
       @param  caller Grid universe from where the acquire invocations have been issued.
       @return        Returns the total acquire hits
*/
AcquireExclusiveHits(Universe source, OID oid, Universe caller)

/** Returns the total acquire misses issued from a universe for a given object from another
universe.
       @param  source Grid universe where the grid shared object resides.
       @param  oid    Object identifier.
       @param  caller Grid universe from where the acquire invocations have been issued.
       @return        Returns the total acquire hits
*/
AcquireMiss(Universe source, OID oid, Universe caller)

/** Returns the total acquire exclusive misses issued from a universe for a given object from
another universe.
       @param  source Grid universe where the grid shared object resides.
       @param  oid    Object identifier.
       @param  caller Grid universe from where the acquire invocations have been issued.
       @return        Returns the total acquire hits
*/
AcquireExclusiveMiss(Universe source, OID oid, Universe caller)
```

The following table presents an example of concrete replication rules, based on some of the above introduced predicates and following the parameter semantics of the *ShallReplicate* method that evaluates one after another the rules within the defined rule table. ***Replication is performed if one of the rules holds true.*** The second column shows the grid object type. The third column specifies the size

of the object among predefined values: small, medium and large, where a mapping from these quantificators to concrete object sizes are provided as system deployment information.

No	Grid Object Type	Size	Condition	Notes
1	READONLY	SMALL	sourceNode != targetNode	
2	READONLY	MEDIUM, LARGE	sourceNode != targetNode && sourceUniverse = targetUniverse && Count(oid, targetUniverse) < IntraUniverseROT	IntraUniverseROT - Intra Universe ReadOnly replication threshold, provided as system configuration
3	READONLY	MEDIUM, LARGE	sourceNode != targetNode && sourceUniverse!= targetUniverse && Count(oid, targetUniverse) < InterUniverseROT	InterUniverseROT - Inter Universe ReadOnly replication threshold, provided as system configuration
4	PRIVATE	N/A	N/A	Private objects are not replicated.
5	MIGRATORY	ALL	sourceNode != targetNode && sourceUniverse != targetUniverse && Count(oid, targetUniverse) = 0	"One per universe" replication rule
6	PC	ALL	sourceNode != targetNode && sourceUniverse != targetUniverse && Count(oid, targetUniverse) < PCT	"n per universe" replication rule, PCT - Producer-Consumer Replication threshold, provided as system configuration
7	READMOSTLY	SMALL, MEDIUM	sourceNode != targetNode && sourceUniverse = targetUniverse && Count(oid, targetUniverse) < IntraUniverseRMT	IntraUniverseRMT - Intra Universe ReadMostly replication threshold, provided as system configuration
8	READMOSTLY	SMALL, MEDIUM	sourceNode != targetNode && sourceUniverse != targetUniverse && Count(oid, targetUniverse) < InterUniverseRMT	InterUniverseRMT - Inter Universe ReadMostly replication threshold, provided as system configuration
9	RESULT	ALL	sourceNode != targetNode && sourceUniverse != targetUniverseD && Count(oid, targetUniverse) = 0	"One per universe" replication rule
10	WRITEMOSTLY	ALL	sourceNode != targetNode && sourceUniverse != targetUniverse && Count(oid, targetUniverse) = 0	"One per universe" replication rule

Table 1: Replication Rules

4.3 Architecture Solution Landscape

In this section we explore several solutions of the system architecture, where we focus on the distributed algorithms, data structures and interaction between nodes and universes. All of the following solutions have in common the problem of implementing mutual exclusion efficiently across the grid. Some of the challenges of mutual exclusion algorithms across the grid are efficiency, scalability and resource consumptions (e.g. number of exchanged messages/consumed network bandwidth). There are three main general approaches in realizing mutual exclusion.

The first approach consists of permission based algorithms such as Lamport [112], Ricart-Agrawala [113], Maekawa [114], which are a class of pessimistic algorithms that work on the principle of consensus among participant nodes before granting access to the shared resource. A second approach are algorithms like Suzuki-Kazami [115], Raymond [116], Naimi-Trehel [117] where a token is passed among nodes and access is granted only to the token owner. Third, optimistic approaches like [118] and [119] work on the principle of transactions and allow resource access and perform a rollback in case of

any conflict. The first class of algorithms suffer from the either the problem of a single point of failure (centralized solutions) or a high message exchange count (broadcasts and consensus based solutions). The second class of algorithms require less messages (typically O(log(N)), but does not ensure fairness, meaning that the order of granting the critical section is not necessarily the order in which the requests have been issued. In other words, resource access grant is not given in the absolute or a Lamport logical clock order, but in the order of the token exchange. The last category of algorithms require additional support for reconciliation in case of conflicting accesses either by the execution run-time or even the programmer. A very recent project that focuses on reconciliation techniques and dependable computing using transactional programming is the Dedisys project [84].

Extensions to the basic algorithms mentioned above have been proposed over the years in [120], [121] where the Naimi-Trehel [117] algorithm is extended in order to reduce the number of exchanged messages and to provide a priority concept. To our knowledge, all of the proposed extension, except [122], do not consider the communication latency between nodes, thus their applicability is limited to homogeneous nodes such as nodes within a cluster. Similar ideas to the ones we have introduced namely grid universe separation and hierarchical structure have been proposed earlier in [123] and [124] where nodes are grouped into groups, but both proposals do not take advantage of the observation that different node groups are connected by larger latency connections. Another important observation is that type consistency was not considered in any of the previous works.

Some of the very recent activities towards better mutual exclusion algorithms on the grid have been elaborated in [122] and [125], where the authors proposed a compositional approach and an extension to the Naimi-Trehel [117] algorithm. Both the proposed extension and the compositional approach could be applied to our universe structure, but there are several required extensions. It is important to note that all of the previously mentioned algorithms refer to the simple case of mutual exclusion. In our work we have to address the entry consistency protocol which requires a different view on mutual exclusion, because in some cases simultaneous access is allowed (e.g. reading via acquire requests). As a consequence, existing algorithms for realizing the mutual exclusion protocol have to be adapted in order to fulfill the entry consistency specification. In addition, type consistency needs to be addressed as well, within the same algorithm, leading to different update mechanisms depending on the grid object type specification.

The architecture of the proposed grid shared data model specified in Chapter 3 requires addressing the following challenges:

- Mutual exclusion algorithm selection/design on the grid scope for entry consistency
- Shared data replication and migration policies
- Entry consistency protocol realization
- Specialized object handling

Pseudo-code notations. We use the following notations in the pseudo-code that describe the algorithms in various solutions that we consider:

```
Synchronous invocations:
// calls the method Method() on the grid node N
Call Method() on N
```

```
// calls the method Method() on the grid node within the list N1,N2…
Call Method() on N1,N2,…

Asynchronous invocations:
// Sends the message identified by Msg to the grid node N and returns immediately
Send Msg to N

// Waits until a message identified by Msg is received
Wait for Msg

// Waits until a message identified by Msg is received from the grid node N
Wait for Msg from N

Message specification:
Msg(A, B, C…) – A,B,C message data members for Msg

Message reception:
// Invoked method when a message Msg carrying A, B, C is received by a node
Receive_Msg(A, B, C)

Timing invocations:
// Performs one or more operations within the prescribed timeout value [ms]
Within timeout do
   Operation1
   Operation2
```

4.3.1 Distributed Centralized Model

A first simple and straightforward solution for the design of the grid shared data layer is the ***distributed centralized*** approach where location and state information is collected and managed in one central location per each universe. In this model, there are *n* nodes in a universe and a special ***"primary node" PN*** which maintains information about the system's run-time status for the universe. A view on the high level architecture is presented in Figure 19 where four connected universes are depicted. We marked with a yellow line the communication flow between universes through the primary nodes. In this case the primary node has a triple role: first, it centralizes status information, it performs access sequencing in the second place and it facilitates all communication between nodes from different universes on the third place. Information and operation requests made by each node are marked with a green line. One can notice that each node contacts the designated primary node within the universe it resides. Data accesses from one process running on a node to the node where data is located are marked with orange arrows.

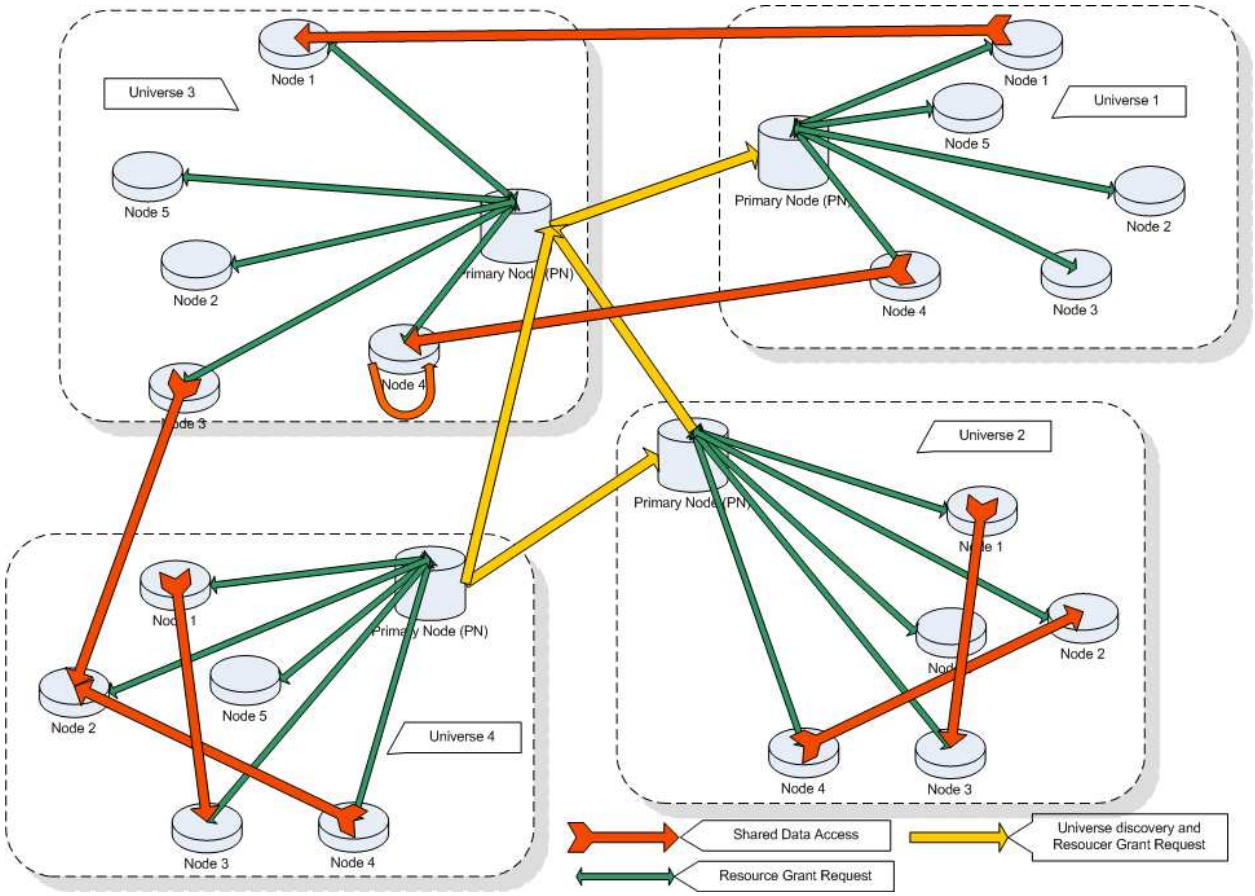

Figure 19: Distributed Centralized Architecture

The rationale for considering this model is simplicity and that it is easy to understand, analyze and implement. One can assume that in case of some particular deployments of the grid universe the number of machines in each universe (and implicitly number of nodes) is limited to a few machines. Larger networks are also deployed, but these cases are not so common on the large scale. It can be assumed that the number of universes in small deployments is likely to be limited to a number from three to five. Considering that inside a universe communication costs are negligible and fast connections are most of the time available, higher communication traffic inside a universe does not have a cost on the network side. It is only the external communication which suffers from large latencies and traffic limitations.

Mutual exclusion handling. Mutual exclusion is realized in this architecture in a distributed centralized way. Each primary node of every universe acts as a central point for sequencing requests to shared grid object for different ranges of object identifiers. Assuming that each universe is associated an unique identifier, *UID*, we define a function *f,* that returns the corresponding universe for a particular grid object identified by its *OID* such as *f(OID) = UID.* The defined function shall follow a uniform distribution, so that requests are fairly distributed among the primary nodes and reduce the possibility of a bottleneck. This situation would be true if replication was fixed, but as in our system the replication mechanism is dynamic, the possibility of bottlenecks cannot be ruled out. Each primary node uses three tables in order to maintain complete information within a universe as described in Figure 20.

- ***UniverseTable*** holds all universes registered in the system. A universe is identified by a system unique id, ***UID***, and the address of the primary node. When a universe is deployed, it contacts the primary node of another universe and registers itself. The registration information is then propagated to all existing universes and the UniverseTable content is synchronized. After each registration, all primary nodes have the same content of the UniverseTable.
- ***NodeTable*** holds the list of currently alive nodes within a universe. A node is identified by a system unique identifier, node identifier, ***NID.*** Each entry of the table contains the idenfier (NID), its address, its total capacity and the current load.
- ***NodeObjectTable*** contains the distribution of shared data objects across a universe. Each entry contains the node identifier (NID), object given and system identifiers (GID, OID) and status information. The dirty flag marks if the object copy is up to date or not. The status flag can one of the following: READY, ACQUIRE, ACQUIRE_EXCLUSIVE, meaning that the object identified by GID is either not held by any process (ready) or acquired exclusively or non-exclusively.

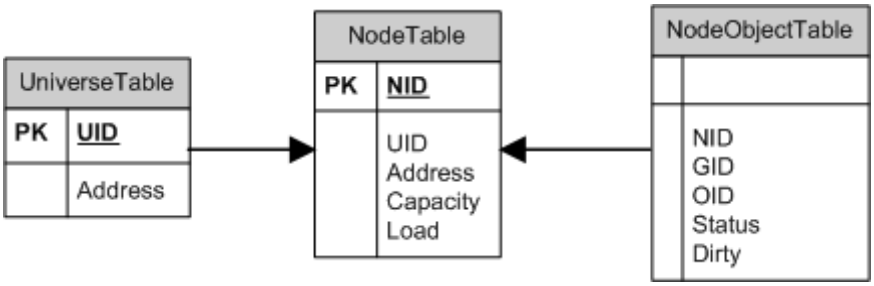

Figure 20**:** Distributed/Centralized Algorithm's Data Structures

The operations performed in this model in oder to fulfill the GridUniverse and GridObject interface specifications and semantics applicable for the *generic grid object* are described in the following pseudo-code. The solution realizes entry consistency specification and update mechanisms via the *write invalidate* protocol. For convenience and conciseness, the pseudo-code of the solution and logic are presented in Appendix A.2. In this model, all operations performed by each node on the GridUniverse and GridObjectRef interfaces delegate to the primary node from the universe the node belongs to. The primary node is fixed, its identity is known by each node within the universe and we assume that the primary node does not fail.

When a primary node receives a *CreateGridObject* request from a node, it tries first to allocate the object on the node the request originates. If this is not possible (e.g. not enough space left on the caller node), a node within the same universe as the node is looked-up. Once identified, a copy of the given object is created on the target node and the primary node updates the NodeTable entries corresponding to the given object. If all the nodes within the universe do not have enough room to accommodate the object, other universes are contacted in order to satisfy the create request. After the object is registered to the primary node where it was created, the object is registered to the designated sequencer. The first registration step assures data locality property by storing universe specific registration information. In the presented code fragment, we try to create the object on other universes sequentially. It would be also possible to send a request to all other universes and delegate the object creation to the first reply received from another universe.

When an object is looked-up by a node, the primary node tries to return first a reference to an object within the same universe. This strategy saves large latency communications to other universes and provides better data locality. If there is no such object within the universe, a lookup request is sent to other connecting universes and the first replied reference is returned to the caller.

When *RemoveGridObject* operation is called with a given identifier, the request is sent to all connected universes. Each universe removes the object if the object's state is *READY*, meaning that the object is not currently used by any process. If the state is not *READY*, each primary node waits until the object is released and its state returns to *READY*.

The strategy for acquire and release operations is that the operation is delegated to the primary node designated to regulate the access to the objects corresponding to the specific object identifier. A lookup function is first applied to the object identifier (OID) and the operation is delegated to the primary node. As a consequence to the function definition, all accesses to the replicas with the same OID are delegated to the same primary node. The concrete execution of acquire and release operations are straightforward, as the designated primary node acts as a global sequencer.

Replication handling. Each time an object is looked-up and it is not found within the same universe as the callerNode, the object could be replicated to the universe the "*FindGridObject*" request is issued from. The decision to replicate or not to replicate depends on several factors. The approach taken in our architecture is to have a replication specification in the form of a rule-based system specified at deployment time and not a built-in decision. In this way it is easy to adjust replication scheme depending on the application requirements. However, due to the fact that locating objects happens on a much lower frequency than object usage, we need to supply replication hooks during acquire operations as well. In this way replication primitive as the number of Acquire or AcquireExclusive operations can be used at their best in order to support better replication decisions. Considering the general design guideline of lazy operations, we could make use of replication hooks in FindGridObject and Release operations.

Preliminary assessment. This solution focuses on centralized information which is concentrated on each primary node. Each primary node holds information about all existing nodes and on all distributed objects belonging to that universe. Also, primary nodes know all other connected primary nodes in the grid universe. The information on existing object instances is created at the time of object creation or replication. Even if the object belongs to one universe, it can be that object access is regulated by a primary node from another universe. It is important to note that even if object existence might be distributed across universes (e.g. during replication), object state appears only to the designated primary node that is mapped to that specific object identifier. Delegation to a single primary node instance ensures fairness, meaning that acquire operations are performed in the order they are issued.

Even if the concept is simple to realize, it has several drawbacks. The obvious drawback is that is has the single point of failure flaw. Second, the architecture is not scalable and the chance for a bottleneck increases with the number of nodes a primary node is mapped to. There is also a problem to maintain a dynamic, optimal mapping of nodes to the primary nodes which in the presented solution is static. Constructing such a mapping is not trivial and would require a global monitoring system which raises some new challenges. Third, local arbitration inside a universe is not possible. Every acquire and release operation results in two calls: one within the universe to contact the local primary node and another remote call to the designated "arbiter" node corresponding to the node identifier. This "arbiter" node might be a potential bottleneck especially if we consider that it might be frequently called over large

latency connections from other universes. An unfortunate situation is depicted in Figure 21, where the grid shared data is located within a universe, but the arbiter node is remote.

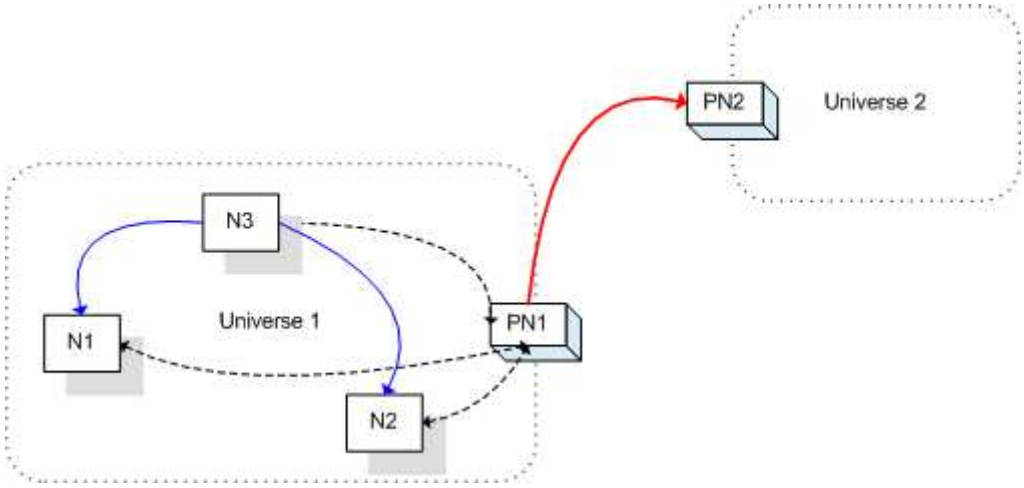

Figure 21: Remote arbiter scenario

In this solution every access to the universe shared data needs to be routed to the far away primary node. A potential alleviation to this problem might be provided by the function mapping definition which shall take advantage of the following observation: when a grid shared data is created we favor the same node or universe that issued the command, thus the mapping function to the object identifier shall map to the same universe where the request is issued from. Unfortunately one cannot totally control replication decisions and application deployment, thus it is always possible that an acquire operation is invoked over a large latency connection to a remote primary node. The frequency of this use cases might be limited by supplying a specialized location function that is tightly connected with the deployment of the application processes across the grid.

In case of broken communication links where disconnected universes are formed, the impossibility to communicate to the designated “arbiter” primary node translates to a complete system malfunction. A solution to this problem would be to promote another primary node for each created partition, with the remark that previous acquire status information is lost. However, we do not focus on fault tolerance and reconciliation techniques, but assume that the universes are connected.

4.3.2 Centralized/Naimi-Trehel Multi-Token Model

This solution aims to alleviate the bottleneck of the previous model by exploring a natural system architectural split and the latency difference between the communication paths within and between universes. The basic idea is to make use of two mutual exclusion algorithms in order to realize entry consistency specifications on the grid scope. The universe structure and node types remains unchanged from the previous solution, and in addition we use a centralized mutual exclusion algorithm within each universe and a token based algorithm between universes (namely between the primary nodes). We considered the Naimi-Trehel [117] token based algorithm as the most promising token based algorithm, due to its higher dynamic behavior and ease of adaptation in comparison to Suzuki-Kazami [115] which relies on multi-casting or Raymond [116]. The solution can be considered as a composition between a centralized algorithm and an adapted version of the Naimi-Trehel [117] token based algorithm. The architecture is similar to the distributed centralized model and is presented in Figure 22.

The idea of the original Naimi-Trehel [117] mutual exclusion algorithm is to make use of a token that is exchanged between processes when a process wants to acquire a resource. Initially all participants know the token owner. When a process that does not have the token requests it, the request follows a tree of probable owners, updating the new owner (the requesting process) as the requests follows the tree nodes. One can imagine that each node is able to get the token by following its own logical tree or probable owners. It is important to note from the beginning that the original algorithm assures mutual exclusion, but does not fulfill entry-consistency semantics as our system specification requires. The original Naimi-Trehel [117] algorithm is described below in pseudo code, where it handles simple locking functionality in contrast to the entry consistency specification that we aim to achieve.

```
// For every node Ni:
Initialization:
        requestinig = false
        next = 0
        if Self = ElectedNode then
                token = true
                owner = 0
        else
                token = false
                owner = ElectedNode
        end if

Acquire:
        requesting = true
        if owner != 0 then
                // the node does not have the token thus the token must be requested
                Send(Request, Ni) to owner
                owner = 0
                Wait for receiving message (Token)
        end if

Release:
        requesting = false
        if next != 0 then
                Send(Token) to next
                token = false
                next = 0
        end if

Receive_Request(Nj):
        // Nj is the requesting node
        if owner = 0 then
                // root node
                if requesting = true then
                        // The node asked for acquire
                        next = Nj
                else
                        // first request to the token since last aquire
                        // send the token directly to the requesting node
                        token = false
                        Send(Token) to Nj
                End if
        else
                // non-root node, forward request
                Send(Request, Nj) to owner
        end if
        owner = Nj

Receive_Token(Nj):
        // receive the token from node Nj
        token = true
```

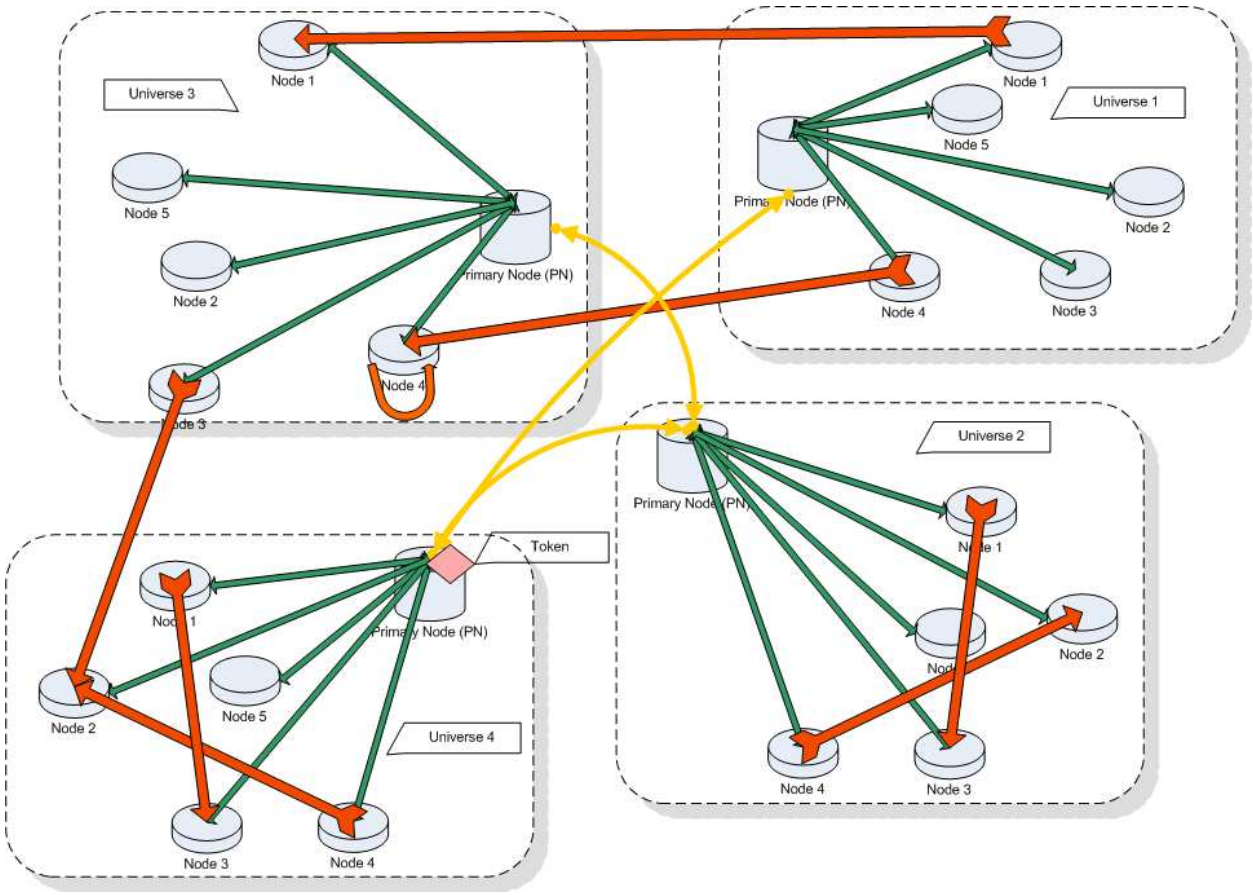

Figure 22: Centralized/Naimi-Trehel Multi-Token Architecture

The rationale for this model is to overcome the bottleneck of the distributed sequencer and reduce uncontrolled large latency calls for each acquire/release operations, by calling only the probable owner chain which limits the message number to *O(log(n)),* where n is the total number of universes, or primary nodes. The authors of [125] concluded that the Naimi-Trehel algorithm is a good choice as a token based algorithm between different distant clusters (in our view, universes), especially if the application pattern has a medium parallelism degree. Besides the good performance and low message exchange count, the dynamic behavior of this algorithm (e.g. maintaining the probable owner tree) makes it easy to adapt and extend. The choice of the centralized algorithm in each universe can be motivated by the fact that, according to [125], the algorithm running inside a cluster (a universe in our view) has a very low impact on the overall performance. There are several choices for the mutual exclusion algorithm inside universes. Besides the centralized algorithm, one can consider token based algorithms or quorum solutions. Considering that a universe is a dynamic and unpredictable structured environment, one needs to address token recovery in the first case or to manage a dynamic quorum as nodes join or leave the universe. In case of quorum based mutual exclusion, one needs to handle the situation where due to slight latency differences and unpredictable delays, the votes are not collected in a predefined time frame and would be sent again. Situations like this increase the complexity of the mutual exclusion algorithm and cannot guarantee a low message exchange count in all situations. All these problems are not raised in case of the centralized approach. Nodes might join and leave the universe at unpredictable times and the message exchange count is always *O(1).* In addition to simplicity of the centralized approach, we

hope that by centralizing information on each primary node it would be easier to apply further extensions that are not so easy to handle on a more complex algorithm.

It is worth to note that the exchanged token in the Naimi-Trehel algorithm becomes a multi-token in our system, having one token per object identifier, like ***token[OID].*** The bottleneck of the primary node can be overcome by requiring that the primary node shall be able to handle in parallel different token types. One mean to realize this requirement is by making use of a multithreaded primary node which would be able to serve requests for different types of objects in parallel (e.g. one thread per each object identifier type). The potential bottleneck of the centralized approach can be further reduced by a more radical solution where the distributed centralized pattern is applied, and divide the responsibility between several nodes within the universe, each node having assigned a range of object identifiers.

Extensions to the original Naimi-Trehel. The first extension is that each primary node holds a request queue, where all requests towards shared objects are stored, and which are issued by nodes from the same universe. The primary node processes the queued requests in FIFO mode, one request at a time. If the token corresponding to the head of the queue is owned by the primary node, the request is satisfied immediately and the requesting node is granted the access to the grid shared data object. If the token is not owned, a request is issued to the probable owner via the Naimi-Trehel algorithm. Requests for different object identifiers, which are handled through different tokens, can be handled in parallel, meaning that the request queue can be seen as a multi-queue, one queue per each token type.

A second extension is considered in case the token is owned by the primary node and another request of the token is received from another primary node. Normally, in this case the token is simply sent after the node from the first universe releases the grid shared object. However, if there are pending requests in the queue for the same token, one can serve those requests instead of the one coming from a remote universe. To ensure a certain degree of fairness and limit starvation, this preemption mechanism can use a preemption threshold. Preemption can be realized both in terms of time and in terms of request counts. In case of “time preemption” we assign a time slice, during which requests are served within the universe instead of remote universes. The “numeric preemption” refers to the situation that if the current preemption is lower than a numerical threshold, a pending request within the primary node’s request queue can be served instead of the remote request.

A third extension is considered for situations where an acquire request is processed and this request is followed by another acquire exclusive or non-exclusive request. While processing the second request, eventually the object’s state must be synchronized. We aim to overlap the second object synchronization with the first object processing by making use of an eager synchronization mechanism for these situations. For these cases we trigger an eager synchronization each time we encounter an acquire request that is followed by another request. Such an optimization can be done by inspection each multi-token queue at the time of handling of each acquire request. All three ideas are depicted in Figure 23.

The operations performed in this model in order to fulfill the GridUniverse and GridObject interface specifications and semantics applicable for the *generic grid object* are described in the following pseudo-code. The requests for acquire and release operations that are issued by each node are sent to the primary node from the universe the node belongs to. The node waits for a reply message within the prescribed timeout values. The interactions are illustrated in Figure 24. For convenience and conciseness, the complete pseudo-code of the solution and logic are presented in Appendix A.3.

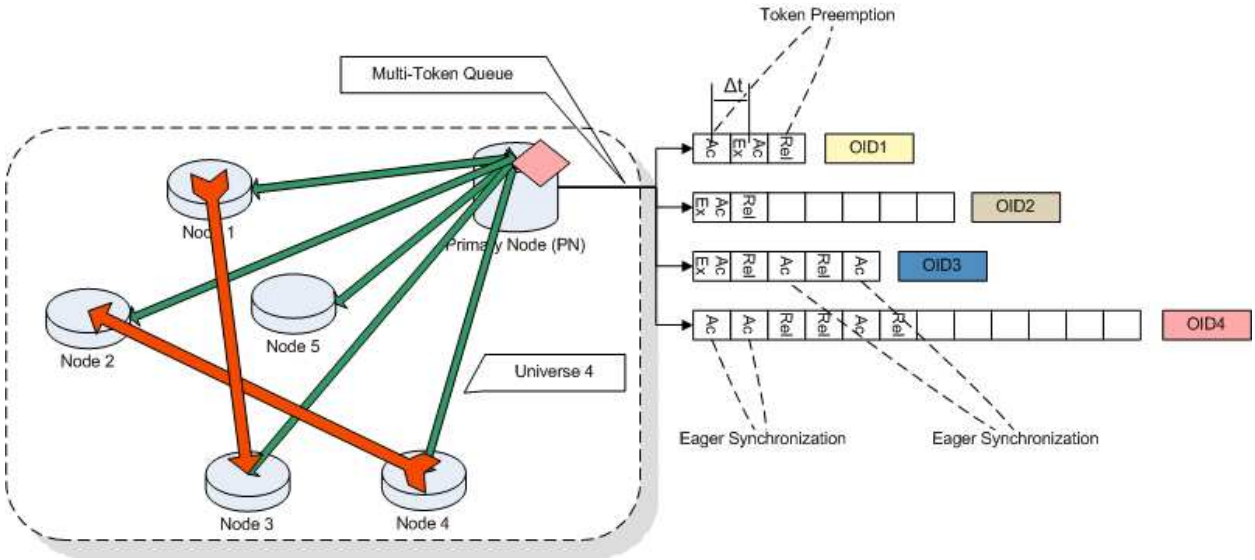

Figure 23: Multi-Token Queue

The node holds the following information:

```
// For every node Ni which has following local variables:
Node : self           - keeps the node identification (Ni)
Node : PN             - keeps the primary node identity (e.g. address)
Boolean : exclusive   - specifies if the previous acquire request was exlusive or not.
```

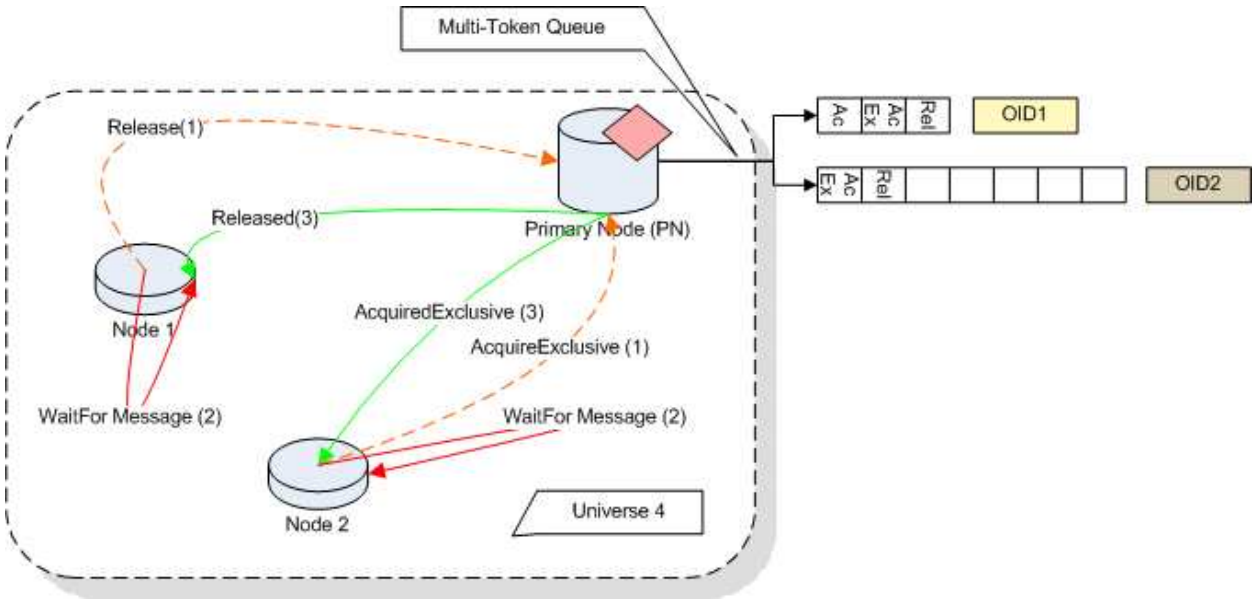

Figure 24: Request-Reply Pattern

The CreateGridObject, FindGridObject, DeleteGridObject and CreateCopy are identical to the ones of the previous model. The other operations on each node of the universes are very similar to those of the distributed centralized model. The nodes simply delegate the operations to the primary node of their universe and in case of the Acquire and AcquireExclusive operations, a reply is awaited in the prescribed timeout value. The data structure for registration information is shown in Figure 25. The same registration and synchronization mechanism is used as in the previous solution. The only difference appears in the NodeObjectTable which does not hold status and dirty flag fields anymore.

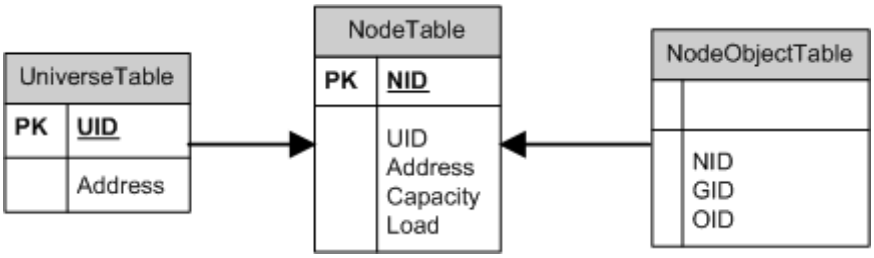

Figure 25: Centralized/Naimi-Trehel Primary Node Data Structure

```
// For every primary node PNi which has following local variables:
Node    : self          - keeps the node's identification (PNi)
Node    : electedNode   - elected initial primary node to hold all tokens
Boolean : requesting[] - specifies if token for an OID has been requested or not
Node    : next[]        - array of next nodes to receive the token for an OID

Struct token{
       Boolean : exclusive    - specifies if the token is owned exclusively
       Node    :  nonex[]     - nodes that hold the token in non-exclusive mode
       Node    :  latest[]    - nodes that hold the most recent copy of an object with OID
}

Token : token[]        - array of tokens for each object identifier
Boolean : owner[]      - specifies if the node is the owner of the token[OID]
Timer   : prTimer[]    - timer for token preeemption for each token[OID]
int     : PREEMPT_TIME – preemption time for token exchange
```

Each primary node keeps a token structure data for each token that corresponds to all object ides that exist in the system. A token contains a Boolean value exclusive that specified if the token is owned exclusively by a node or not. It contains also a list of all nodes that are currently accessing the data in non-exclusive mode and a list of latest nodes that contain a copy of the latest value of the object referred by the object identifier OID.

The CreateGridObject and DeleteGridObject operations are similar to those of the previous model, with an important and subtle change: upon registration or replication, the grid shared data object is not registered anymore to the global designated sequencer, but to the primary node where it belongs to. Thus, the information about an object or one of its replicas appears only in one universe. The FindGridObject method is identical to the one of the previous model and for the sake of space is listed only in the appendix A.3.

Both Acquire and AcquireExlusive operations add a request to the processing queue of the primary node. The primary node takes each request from the head of the queue, decodes the request type and delegates to the required operation handler (PN_Acquire or PN_AcquireExclusive). The interactions for Acquire operations are described below and depicted in Figure 26 and Figure 27.

There are a few more extensions to the original Naimi-Trehel algorithm that can be seen in the above algorithm illustrated in pseudo-code. The first extension is to differentiate the two acquire operation types. Second, when a node request non-exclusive acquire, the token is not requested, but a message is sent to the token owner. If the token is not already in exclusive mode, the token stores the requesting non-exclusive node in a list. All the nodes in the list could own the token in non-exclusive mode without having the token. When the node releases the resource in non-exclusive mode, the probable owner path is followed again and the node is removed from the list. The operations in the AcquireExclusive case are similar to the original Naimi-Trehel algorithm, with the exception that the exclusive mode is granted only if the token's node list is empty, meaning that there are no current nodes in non-exclusive mode.

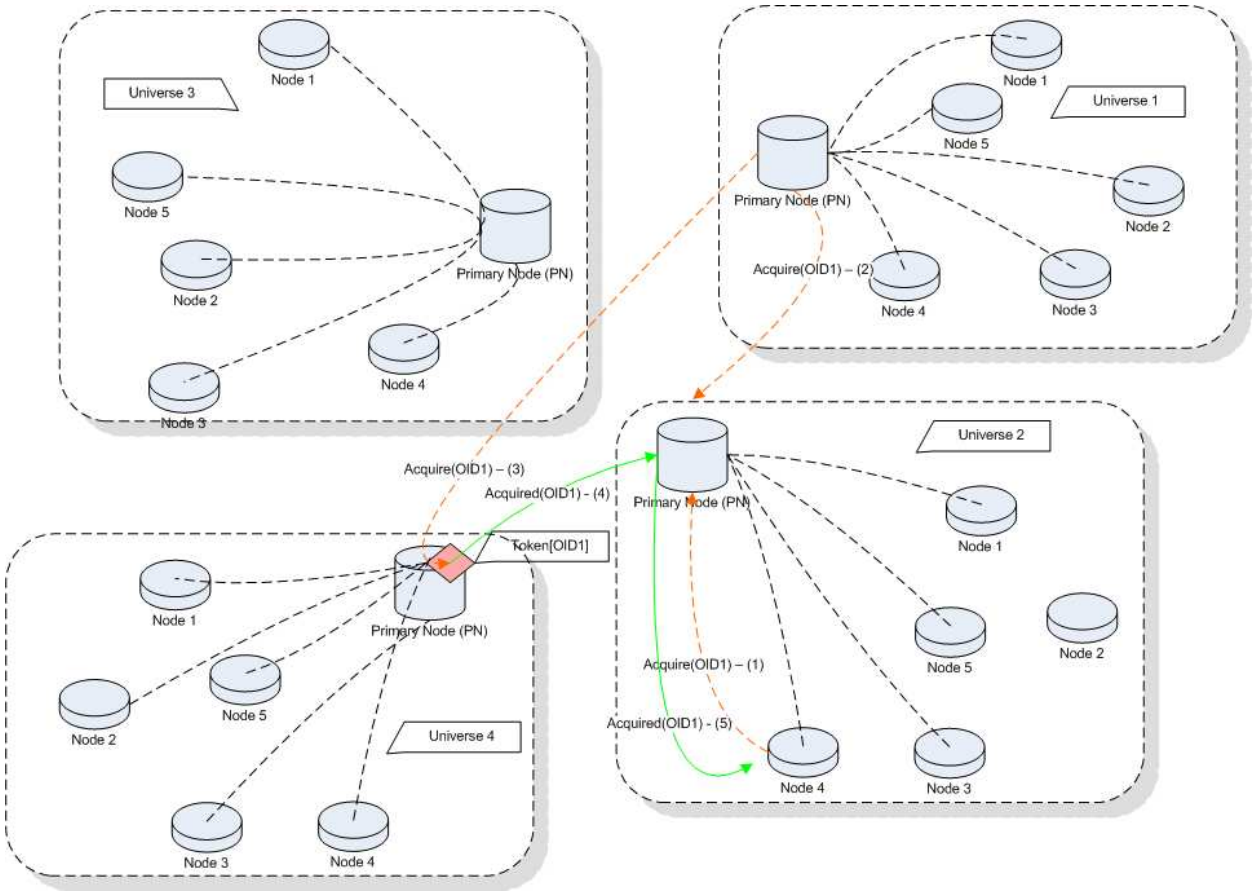

Figure 26: Acquire Interactions

In order to fulfill entry consistency specification, each token has been extended with a list of nodes (*token[OID].latest*) which keeps track of the nodes that hold the most recent copy of an grid shared object which has the type OID. Initially, when the first object is created, the node where the object is created is added in to the *token[OID].latest* list (*PN_Initialize* method). When an object is acquired in non-exclusive mode, the AcquireGranted message transmits the latest node list too. If the node that holds the object pointed by the reference is not part of Latest, the object is synchronized using one of the replicas from the Latest list. Object synchronization is described in the next chapter that focuses on the detailed design. When an object is acquired exclusively, the same synchronization mechanism applies. In addition, after the synchronization operation, the Latest node list is reinitialized with the current node that issued the *AcquireExclusive* request.

Preliminary assessment. Compared to the previous solution, an acquire request requires a message to the primary node of the same universe as the caller. If the token for the object identifier is not owned by the primary node, a number of *O(log(n))* calls might follow, in order to locate the current owner, where n is the total universe count, or the primary node count. Compared to the previous solution, the possibility of a bottleneck is alleviated because each primary node is able to serve requests in parallel per each object identifier group (OID). In the situation that the token is already owned, the solution has the advantage or a very fast “Acquire” operation because it is resolved within the universe on a low latency connection. Although retrieving the token requires more than one message, as in the distributed-centralized approach, this algorithm provides a better scalability, thus a higher number of universes could

be connected in this solution. Applying the preemption idea can lead to a better token locality and saving large latency communication calls especially in “trashing scenarios” where the token is asked repeatedly from one or more distinct universes.

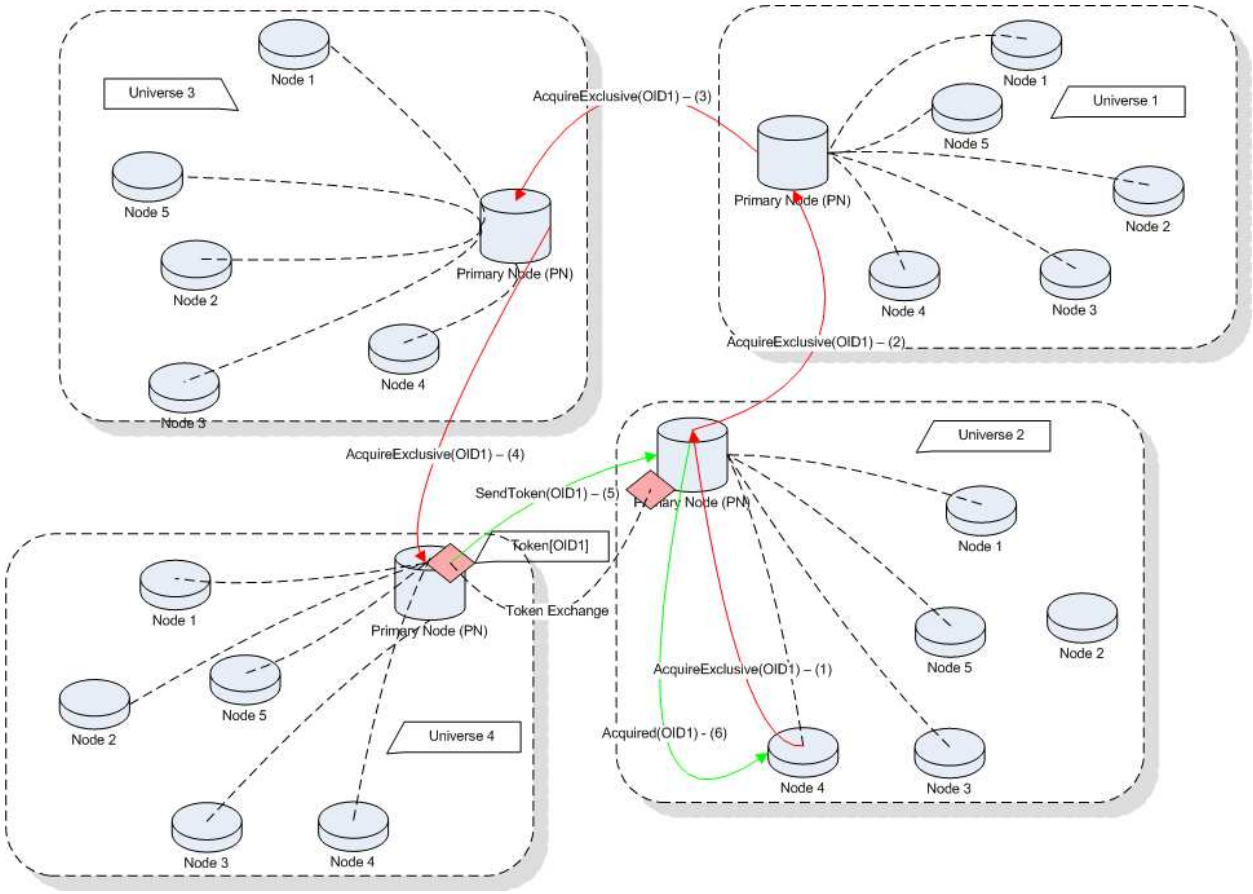

Figure 27: Acquire Exclusive Interactions

In case of broken communication links where network partitions occur, the algorithm could work independently per universe through the centralized algorithm part, assuming that the primary nodes do not fail, but only the communication links are broken. Such a scenario is not possible in case of the previous solution, where a remote universe is always contacted to perform sequence the requests. Of course, if a primary node fails, all universe-related object registration is lost, together with the currently held tokens. System rehabilitation could be accomplished through leader election algorithms and promoting regular nodes to primary nodes.

From the possible system deployment scenarios, the most promising deployments where this solution fits best would be the ones consisting of universes mapped to a medium machine count (10-20) and variable number of universes. From the application parallelism level, we expect that a highly or medium parallel application would work best in this setup. A low parallel application that has frequent synchronization points would probably trigger frequent token exchanges over large latency connections and would probably have a weaker performance. Although the type consistency object specification could help in this respect, it is hard to judge to what extent would make a difference on a general basis. We expect that a proper object type choice in case of low parallel applications has the highest impact on

overall performance. As the application exhibits increasing levels of parallelism, the impact of the object type choice would probably decrease.

4.3.3 Hierarchical Models

In this section we explore other potential architectural solutions for our system specification based on very recent community research results. The first class of solutions is obtained through an algorithm composition approach. In this case we considered different compositions of mutual exclusion token based algorithms on two levels: one level within each universe and another level between universes. An analysis of the performance and suitability of different token based algorithms has been presented in [125], where the authors have evaluated the following combinations against the Naimi-Trehel algorithm applied on the global grid scope:

1) Naimi-Trehel/Martin
2) Naimi-Trehel/Naimi-Trehel
3) Naimi-Trehel/Suzuki-Kasami

The original Naimi-Trehel algorithm is not suitable for a widely distributed system as the token is retrieved by following multiple branches in the logical “probable owner tree” that could span across large latency communication channels. For this case, the architecture is depicted in Figure 28.

The authors of [125] concluded that the Naimi-Trehel algorithm is the most suitable for inter-cluster token handling and the choice of the intra-cluster algorithm has less impact on overall performance. While the third combination (Naimi-Trehel/Suzuki-Kasami) has the least obtaining average time, it has the highest total sent inter-cluster messages and thus higher bandwidth consumption. On the opposite side, the first combination (Naimi-Trehel/Martin) has the highest average obtaining time, but the least number of exchange messages between clusters. In terms of suitability for different types of applications, in case of low parallel applications, Martin’s algorithm has the same obtaining time as Suzuki-Kasami, but sends far less messages, thus being a good choice in this case. For intermediate parallel applications, Naimi-Trehel exhibits a similar obtaining time as Suzuki-Kasami, but with the advantage of less exchanged messages. For highly parallel applications, Suzuki-Kasami is suggested as providing the shorter obtaining time with an increased number of exchanged messages due to the frequent broadcasting scheme. A solution that extends the one level token model, by applying the Naimi-Trehel token based algorithm at two levels has been presented in [122]. Applying the concept in our model, in this approach a unique token exists per grid universe and object identifier. In this way the bottleneck of the primary node as in the distributed-centralized model is removed, but the number of messages is increased as the probable owner path to the token is longer. The authors of [122] have extended the original algorithm by applying the following extensions:

1) A proxy node is introduced per cluster that routes all requests to other clusters. The purpose is to cache the token’s owner and route a request directly to the owner if the owner is located in the same cluster.
2) An aggregation concept is used so that remote requests are aggregated and only sent to remote clusters if there have been accumulated a certain number of requests.
3) Token preemption is performed so that requests within a cluster are satisfied before remote requests if the preemption count does not exceed a certain threshold.

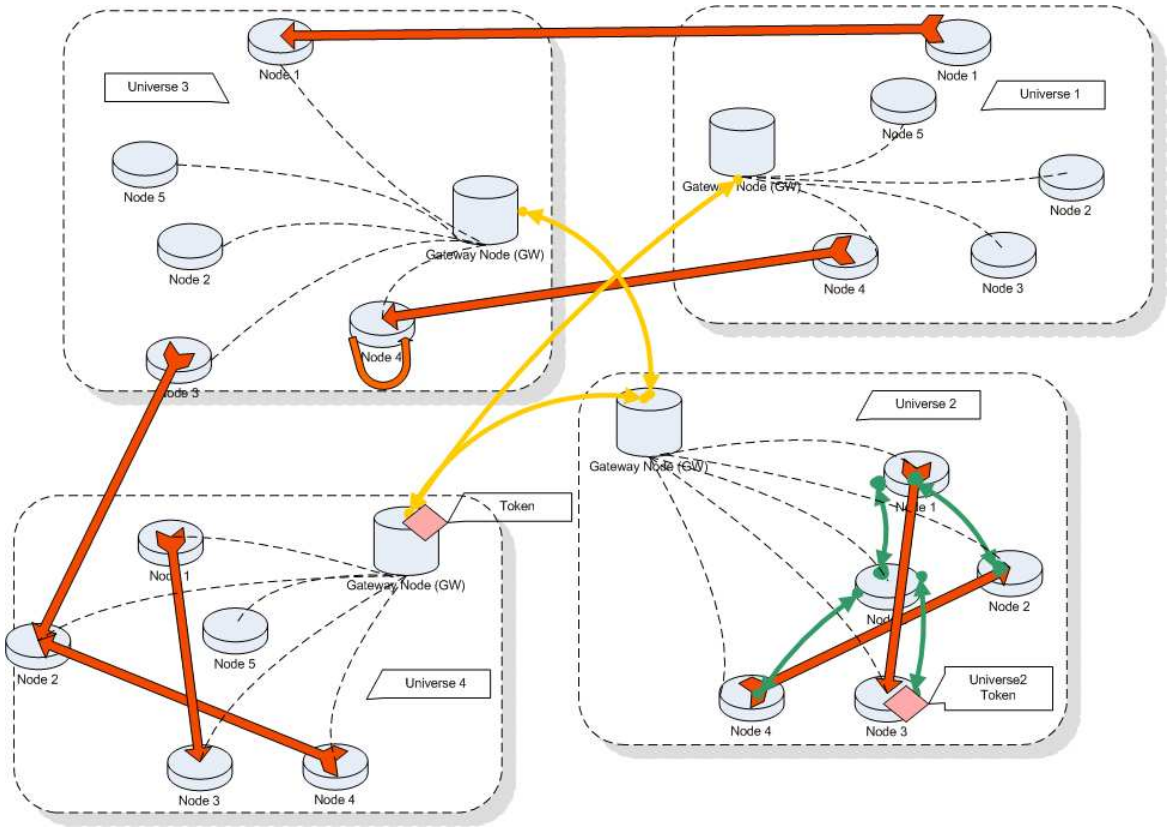

Figure 28: Hierarchical Naimi-Trehel multi-token architecture

Preliminary assessment. The authors of [125] concluded that it is only the “between universes” algorithm which brings a significant performance impact on the system, whereas the algorithm applied inside a universe has no major performance impact, except in the number of exchanged messages. Although the experiments run on [122] were conducted on a grid made out of three clusters each of three machines, it is clear that the original Naimi-Trehel algorithm is superior to the centralized and broadcast algorithms in terms of token obtaining time. Applying the preemption concept and the implicit first extension to the original Naimi-Trehel led to the most efficient solution in terms of the obtaining time. Based on the measurements in [122], it seems that the Naimi-Trehel algorithm is the most suitable for exchanging tokens between universes and it provides a reasonable trade-off between different classes or applications (highly parallel vs. low parallel applications). It is not clear how the enhanced Naimi-Trehel algorithm relates to other three compositional solutions. What we can assume is that it exhibits a similar behavior as the solution presented in 4.3.2. All optimizations except the aggregation concept (which did not prove quite useful) can be also seen in the Centralized/Naimi-Trehel multi-token algorithm. Keeping in mind that the experiments were not conducted on a large latency wide distributed grid, but on a 100MBit and 10GBit connections, the results might be unknown in an environment dominated by large latency connections.

Considering the implementation aspect, compositional approaches are easier to understand and implement. The extended Naimi-Trehel algorithm is the most complex solution. While none of those solutions address the entry-consistency specification, but only mutual exclusion, adapting the extended Naimi-Trehel algorithm requires a considerable effort. In addition, one must consider that there are also extensions required in order to handle the different kind of object types.

4.4 Solution Selection

In the previous sections we have identified six potential candidate solutions for realizing mutual exclusion with entry-consistency semantics in a grid environment. We could add transactional memory approaches as well, but these are more suitable for tightly-connected nodes. To our knowledge, there have been several attempts towards grid transactional services specifications, which are basically an extension to grid service specification, but no realization of such theoretical models for large scale distributed systems. Second, transactional programming implies different semantics to our system specifications and requires reconciliation techniques when transactions fail. Most of the times, transactional programming specifies sequential consistency. According to [126] and [127] there are very subtle and complex aspects when relaxed consistency models are specified for transactions. Unfortunately, it is hard to foreseen all challenges of this approach as there is very little research evidence of relaxed consistency transactional programming. As a consequence we do not consider transactional models in the solution selection debate.

The first out of the six solutions described in 4.3.1 has only the advantage of simplicity. The disadvantages of the sequencer bottleneck, inability to arbiter requests locally and the single point of failure rule out this solution immediately among the candidates. The second solution presented in 4.3.2 combines both scalability and locality features on one hand and removes the disadvantages of the previous solution. Thus, it could be a candidate and we keep it for further comparison. The next three solutions are the three compositional approaches described in 4.3.3. These are similar to the previous solution, the only difference is the mutual exclusion algorithm applied inside a universe. The last choice is the adapted Naimi-Trehel algorithm described in [122] which is applied on the grid scope. This solution requires a “gateway node” in each cluster in order to keep track if the token is held remotely or not. From this point of view, this design approach resembles to the solution presented in 4.3.2.

Based on the measurements in [122], it appears that the Naimi-Trehel algorithm is the most suitable for exchanging tokens between universes and it provides a reasonable trade-off between different classes or applications (highly parallel vs. low parallel applications). Following this observation, a reasonable solution choice is among those which rely on this particular algorithm for mutual exclusion between universes. In the following table, we summarize the characteristics of each of the four remaining candidates, where we highlight the negative characteristics of each solution by marking them in bold/red style. In Table 2 each assessed characteristic is rated using three values: low, medium and high. The meaning of these values is specific to the given characteristic and is described in Table 3.

Criteria	Centralized/N-T	Martin/N-T	Suzuki-Kasami/N/T	Grid N-T
Universe scalability	high	high	high	high
Local scalability	**low/medium**	**low**	medium	high
Local obtaining time	**low/medium**	medium/high	medium	low
Local resource demand	low	**medium**	**high**	**low/medium**
Independent processing	high	**low**	**low**	**low**
Complexity	low	**medium**	**medium**	**high**
Local dynamics	high	**low/medium**	**low/medium**	low

Table 2: Solution Criteria

Universe scalability refers to the number of connected universes which can be supported by each solution without significant performance issues. All solutions present the same universe scalability, meaning that we expect the same behavior when the universe number increases. Local scalability refers

to scalability inside universes due to the increasing number of nodes. Although it relies on a central resource, due to the ability to handle parallel requests per object identifier, the first solution is expected to have a low to medium scalability. In case of the Martin algorithm, as the logical ring length increase, the scalability factor decreases. The next two solutions have higher scalability due to the missing centralized resource or the dependency on the number of nodes. The local obtaining time refers to the time elapsed from the moment the acquire operation is issued, until the access to the shared data is granted. In case of the first and last algorithms we expect a low to medium value, considering the ability of the centralized algorithm to handle parallel requests. The Martin and Suzuki-Kasami algorithms are expected to have increasing obtaining times due to their dependencies to the number of participant nodes

Criteria	Low	Medium	High
Universe scalability	up to 5 universes	5 to 15	15 to 50
Local scalability	up to 10 machines	5 to 50	50 to 100
Local obtaining time	10-10ms	100-500ms	500-5000ms

Table 3: Solution Criteria Description

Local resource demand translates to the number of exchanged messages in order to acquire a resource. The first algorithm requires only *O(1)* messages to the primary node. The second algorithm requires to traverse the entire logical ring, leading to *O(n)* messages. The same is valid to the third algorithm which relies on broadcasting, thus it sends and expects n-1 messages, which means a performance of *O(n).* In case of the last algorithm, it has only to traverse the local logical tree, which in the worst case requires *O(log(n))* messages and *O(1)* in the best case where the location of the token is cached. The independent processing ability refers to failure scenarios where communication between universes is broken. When universes are disconnected, but the machines within universes are functioning properly, the first three algorithms enable the system to continue independently as the algorithms rely only on universe specific information. The global Naimi-Trehel algorithm requires locating the token which might belong to other universes, thus its proper operation is seriously hindered.

We consider by “local dynamics” the normal universe evolution where machines are becoming available or disappear from the universe. This situation is more frequent than “universe dynamics” as we assumed that the primary nodes are always available. In this case, the first solution is not impacted, as it makes use of local information only and there is no distributed information stored on any of the nodes within a universe. The following two solutions require the reconstruction of the logical ring and the multicasting group, which leads to lower dynamic characteristics. Although adding new machines in the last solution is easy to handle, removing a machines that has a token results in system failure. Leader election algorithms are required to deal with this kind of conditions. Last but not least, in terms of complexity, the first solution is easier to realize than any of the other solutions. The adapted global Naimi-Trehel algorithm is the most complex solution due to its distributed data structures and additional token-related information synchronization protocol.

Considering all the above factors we chose for our system specification the ***Centralized/Naimi-Trehel Multi-Token*** algorithm [128]. The core motivation for this choice is its high local dynamics, higher capability to perform independently, a low resource demand and low complexity. We have traded the local scalability for all other characteristics as we believe that universes will have a limited number of nodes (up to 20) for typical deployment scenarios. Last but not least, we believe that this solution allows us to realize object specific handling in a more efficient and maintainable way and the bottleneck of the primary node will be greatly reduced due to parallel processing of object type requests.

5 Detailed System Architecture

In the previous chapter we have presented the solution for realizing mutual exclusion on the grid scope and the way entry consistency is realized for the generic grid object type. In this chapter we address the design for specialized grid objects and define in more details the node deployment and object access mechanism. Universe nodes can be deployed in any number, on any grid machine. A single node deployment scenario is depicted in Figure 29 where one universe node is deployed on a grid machine where two applications are running. In this case the applications that are running in their own processes access the universe node's services via an inter-process communication mechanism (IPC).

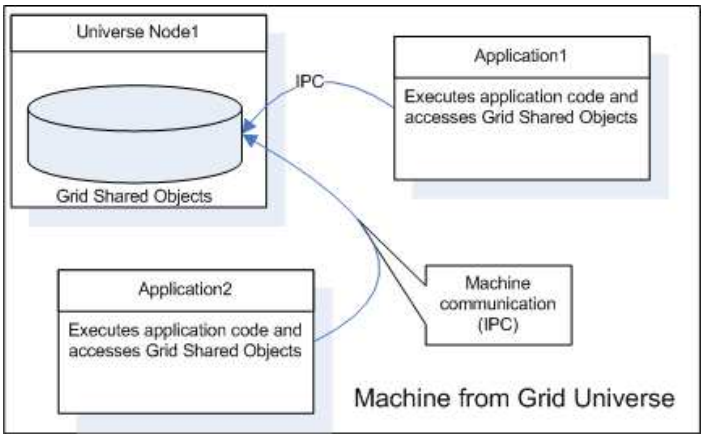

Figure 29: Single Universe Node Deployment

A multiple universe node deployment scenario is illustrated in Figure 30, where two applications are running in their own processes and two universe nodes are deployed. Both applications are communicating to each other by accesses data from both universe nodes.

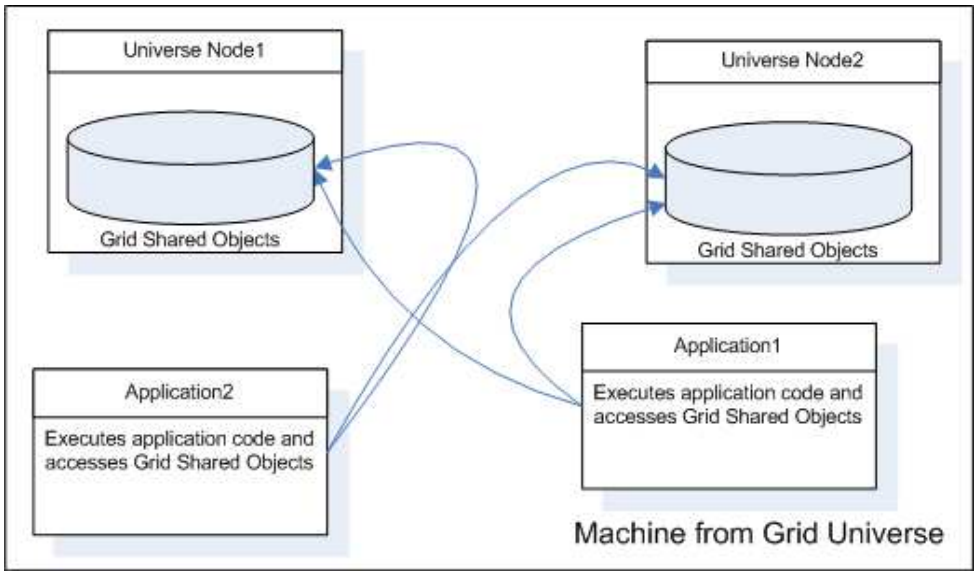

Figure 30: Multiple Universe Node Deployment

The deployment process implies assigning universe nodes on a set of grid machines *S1* and spawning application codes across another set of machines *S2*. Interactions between universe nodes and application process happen either within the same machine or remote, across machine boundaries, as depicted in Figure 31. *We require that on each machine where an application process is launched, there shall be at least one universe node deployed*. In other words S2 <= S1. The rationale for this consideration, and therefore a system deployment constraint, is that we aim to provide better data locality. In case there is no universe node deployed on a grid machine, and an application is running there, there is no chance to replicate data on the same machine as the universe node does not exist on that machine. However, if universe nodes exist on all machines where computations are running, there are better chances to replicate data on the universe node located at the same grid machine and thus to favor local data access over remote data access.

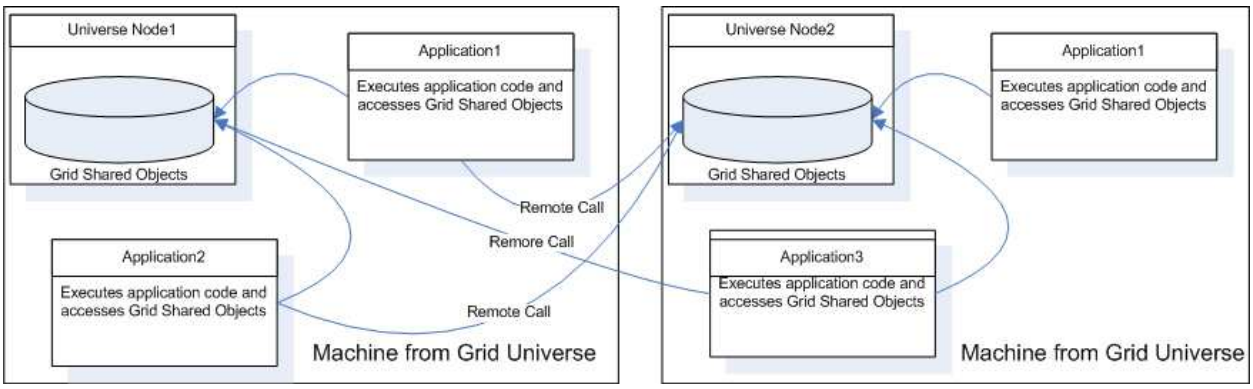

Figure 31: Single Node Multiple Machine Interactions

Every interaction between a grid shared object and the application process happens over a reference defined previously as GridObjectRef. A reference is obtained when an object is created or looked-up and is used to manipulate the referred object's state and to coordinate access status by acquiring and releasing the referred object. In our model, the reference holds internally the following information:

- OID - object identifier assigned by the programmer/user
- GID - global unique identifier assigned by the system
- DataNode - identity of the node where the referred data is located and where all accesses are redirected (e.g. acquire/release and payload read/write)
- Client - identity of the calling process (client) that is currently manipulating the referred data.

It is important to note that every calling client that uses a reference is having its own reference that is constructed by the system and given on the client space. Some of the reference's internal data parts are fixed like the OID and the GID whereas DataNode and Client are dynamic. Some of the parts are always available and known like the OID, GID, but the others can be known only in some stages of the execution. When the reference is created for the first time all four parts are known. For instance, when the referred data item migrates from one machine to another, the DataNode becomes invalid for the duration of the migration process. As a result, after a data item migrates to another node, the DataNode value might point to the incorrect node. In order to access the correct node, the access request must be routed among the probable owner tree of each node, as described in the Centralized/Naimi-Trehel

algorithm. Thus, all processes that hold a reference to a migrating data item might experience the situation that the DataNode is invalid at some point in time. We say that when all four parts of the reference is known and valid that the *reference is resolved*. When at least one of the variable parts is not known or invalid, we say that the *reference is not resolved*. A sample grid reference is shown in Figure 32.

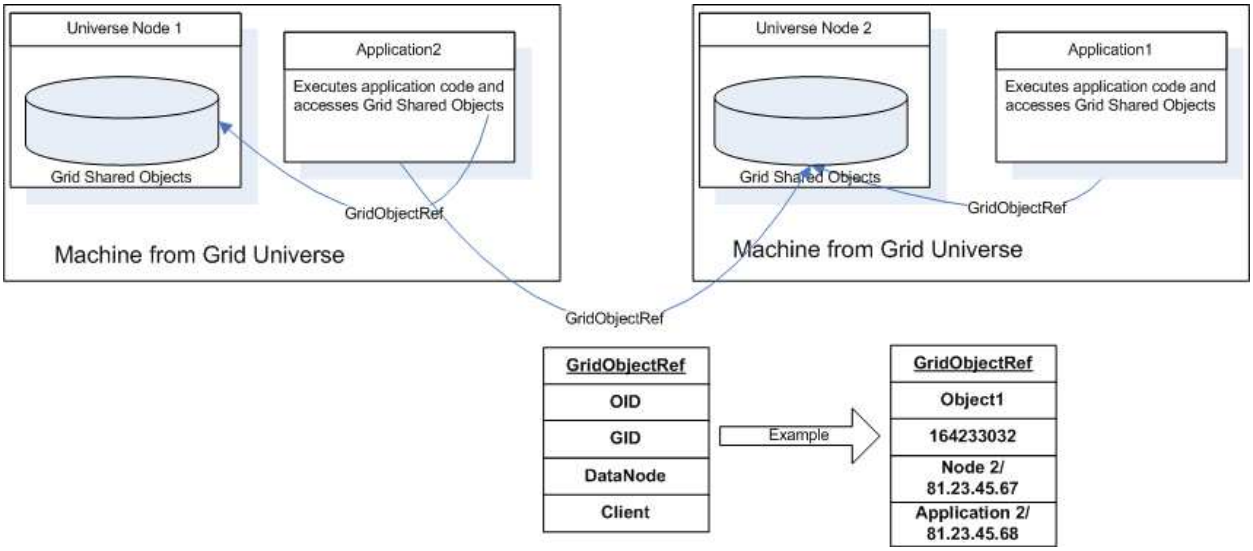

Figure 32: GridObjectRef Internal Structure

5.1 Read-Only Objects Handling

Read-only objects were defined in *Definition 3.12* as immutable objects which do no propagate any potential state change to any of their replicas. As a consequence it would be beneficial to replicate them with a high rate so that read accesses could benefit of the data locality property and increase the chances for local accesses instead of remote accesses. The main purpose for read-only objects is to reduce access time due to large latencies. The default replication rules defined in our system in section 4.2 ensure that read-only objects are highly replicate as close as possible to the location of the process that requests them, which are referred as caller nodes. It is important to note that read-only objects and are not migrated, but only replicated. The overall characteristics of read-only objects are summarized in Table 4. A visual representation of read-only objects handling is shown in Figure 33.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Read-Only	n	yes	no	n	0

Table 4: Read-Only Object Characteristics

In the scenario depicted in Figure 33 we created three different read-only objects that are used by one application running on the same machines where the universe nodes are deployed. Read-only objects ReadOnly1 is replicated on all machines, but the other two, ReadOnly2 and ReadOnly3, are not replicated yet (we could consider that the object is quite heavy and replication rules did not evaluate to true yet). One can observe that applications are accessing read-only objects residing in their proximity whenever possible (e.g. a replica within in their proximity): ReadOnly1 object is always accessed from the same grid machine instead of accessing a remote replica over large latency connections.

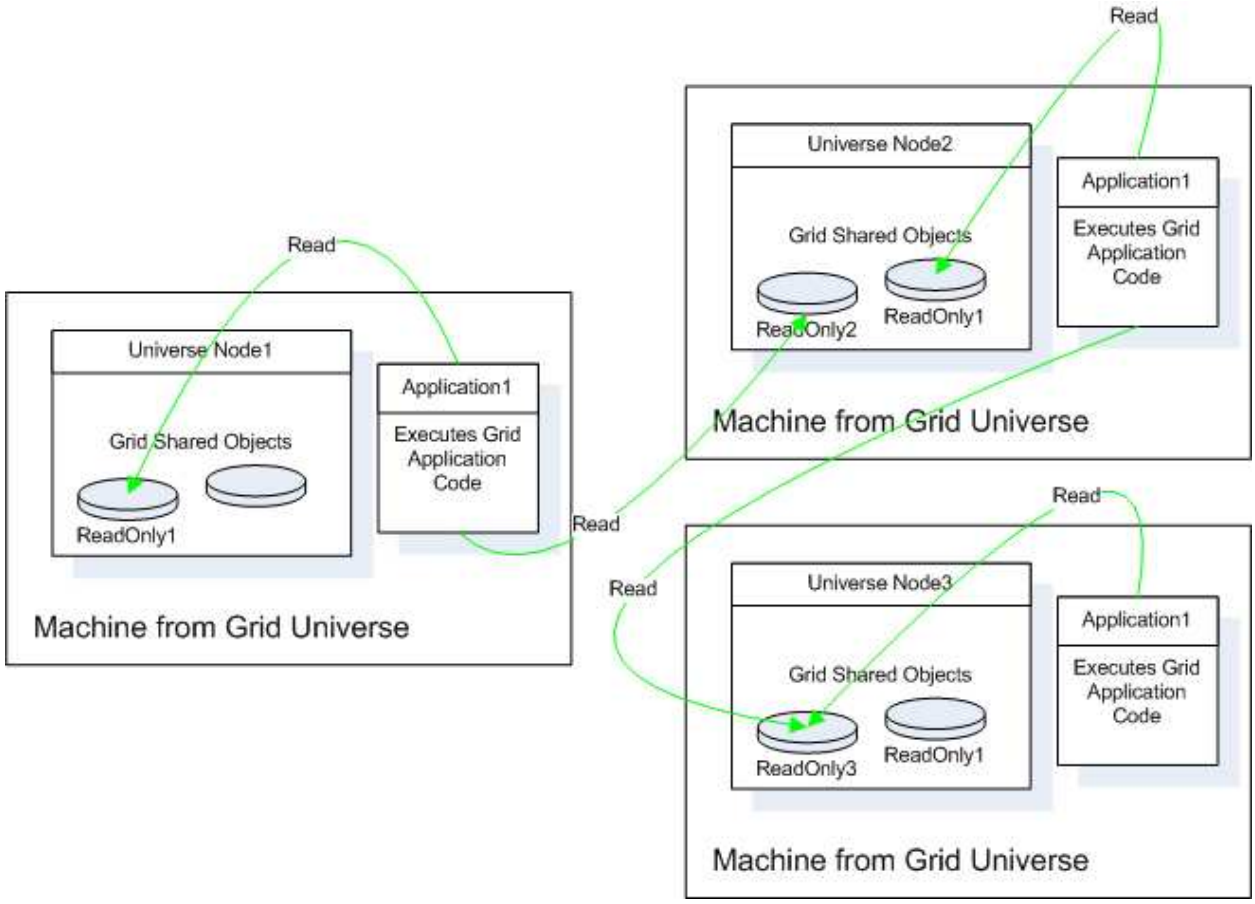

Figure 33: Read-Only Objects Handling

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

As the payload or state of a read-only object is immutable, such objects do not require any locking mechanism. As no arbitration or token request is required anymore, a “do-nothing” logic is provided in the Acquire and Release methods for each node, Ni as following:

```
// For every node Ni
/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref           Grid object reference.
       @param  timeOut       Timeout value.
       @return               True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut)
       if (Type(ref) == READONLY)
              return true
       end if
       // Code continues from the generic grid object handling

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref           Grid object reference.
       @param  timeOut       Timeout value.
       @return               True if successful, otherwise false. */
```

```
AcquireExclusive(GridObjectRef ref, long timeOut)
        if (Type(ref) == READONLY)
                return true
        end if
        // Code continues from the generic grid object handling
        . . .

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference, OID.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref)
        if (Type(ref) == READONLY)
                return true
        end if
        // Code continues from the generic grid object handling
        . . .
```

Consequences. The following consequence apply for grid read-only objects: Changing the state of a read-only object is syntactically possible, but represents a semantics violation (programming error).

As an example, if a read-only object's state is changed, its new state is not synchronized to any of its existing replicas. In the situation that a read-only object's state is changed, the state of any replica that is created after this point in time is undefined (initial value or the changed value). The same applies to the find operation when a reference to such an object is returned, what might point to an arbitrary read-only object (the correct object or the modified object).

5.2 Private Objects Handling

Private objects introduced in *Definition 3.13* are bound to a fixed location and are typically used by one or more local processes belonging to the same grid machine. Normally, private objects are used only locally, but from the syntactical point of view they can be still accessed remotely. As a consequence, arbitration can be handled on the local scope instead of the grid scope. The main reasons for private object use are situations where an object is shared locally, by multiple processes within the same machine or machines within a low latency connected universe. In this case, a locking mechanism in the grid scope is avoided, thus a higher performance is expected for mutual exclusion in the machine scope. Typically, there is only one copy of a given private object instance, meaning that private objects are not replicated. As the location is bound to a certain node, private objects are not migrated either. The overall characteristics of private objects are summarized in Table 5. A visual representation of private objects handling is shown in Figure 34.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Private	1	no	no	n (local)	n (local)

Table 5: Private Object Characteristics

In the scenario depicted in Figure 34 we consider that the application is using private objects from a point in time when the computation logic do not require any information exchange between remote application processes. Each application process uses private data located in the same grid machine. However, remote access is semantically possible, but strongly discouraged as we will point out this below.

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and

conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

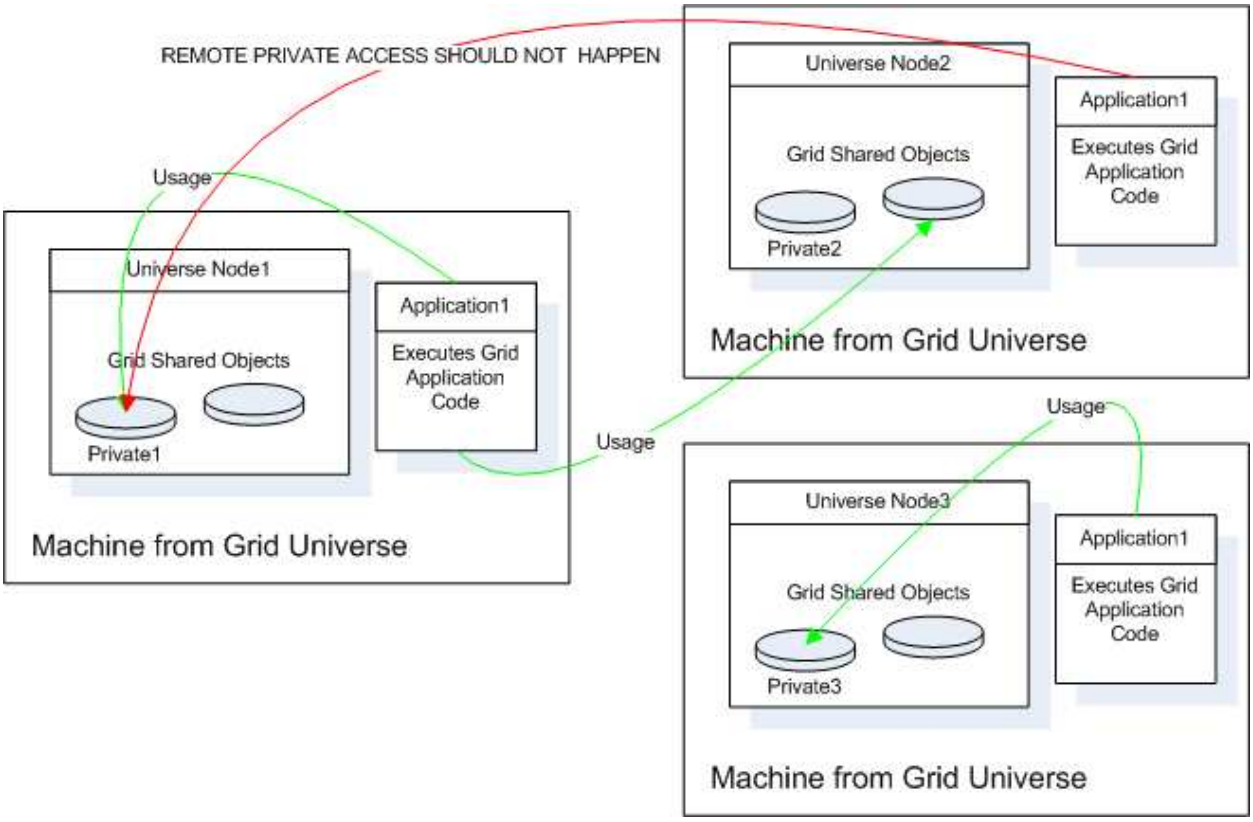

Figure 34: Private Objects Handling

Private objects require a machine specific locking mechanism which can be handled in the arbitration methods on each node, without requiring any mutual exclusion algorithm between primary nodes. Instead of the “do nothing” implementation of read-only objects, we make use here of a read/write mutex object that performs a process specific read and write lock according to the entry consistency specification. Each universe node that contains a private grid object has a read-write mutex associated with each object type instance. Modern operating systems like WindowsXP or runtime executive systems such as Java (starting with version 1.5) provide natively such construct. Such locking object can be implemented using a standard mutex and a semaphore object. A sample implementation of such an object is listed in Appendix A.1.

```
// For evey node Ni
RWMutex[]       – Read/Write mutex that protects a resource for read/write access according to
                entry consistency semantics, one per each object type

/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut)
        if (Type(ref) == PRIVATE)
                return RWMutex[ref.OID].ReadLock()
        end if
        // Code continues from the generic grid object handling
```

```
/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref           Grid object reference.
       @param  timeOut       Timeout value.
       @return               True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut)
       if (Type(ref) == PRIVATE)
              return RWMutex[ref.OID].WriteLock()
       end if
       // Code continues from the generic grid object handling
       . . .

/** Releases a previously acquired grid object referred by a grid object reference.
       @param  ref           Grid object reference, OID.
       @return               True if successful, otherwise false.
*/
Release(GridObjectRef ref)
       if (Type(ref) == PRIVATE)
              return RWMutex[ref.OID].ReleaseLock()
       end if
       // Code continues from the generic grid object handling
       . . .
```

Consequences. The following consequence applies for grid private objects: Private objects are looked-up on the grid scope, but arbitration is done on local, machine scope.

5.3 Migratory Objects Handling

According to *Definition 3.14* migratory objects are accessed in phases. In each phase different processes are taking exclusive access to the given object and then releasing the object. A migratory object is not replicated, thus, a single copy exists in the grid universe. As only one process is accessing a migratory object, a mutual exclusion protocol on the grid scope is not required. Normally the object is used exclusively by one process that resides in the proximity of the node where the migratory object has been migrated. However, the syntax of the interface allows the object to be read from a remote process, but the reading request is granted after the object has been released from the previous AcquireExclusive operation that triggered the migration process. The main benefit of this object type can be observed in cases where an object is used for a short, but intense period of time when data locality is important and replication would not bring a significant performance gain. The overall characteristics of migratory objects are summarized in Table 6. A visual representation of migratory objects handling is shown in Figure 35.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Migratory	1	no	yes	n	1

Table 6: Migratory Object Characteristics

In the above scenario, a process of Application1 is creating a migratory object on the UniverseNode1, on the same machine where the application code is running. The process is making use of the object by reading and writing data to it (1). In the next step, another process of the same application located in a different grid machine locates the migratory object and calls AcquireExclusive (2). When the first process finishes its operations with the object and releases the reference, the object migrates to the universe node in the proximity of the second caller (3). The second application process starts using the object (4). Next, the third application process gets a reference to the same migratory object, calls AcquireExclusive (5) and waits until the access is granted. When the second application process releases the referred object, the object migrates to UniverseNode3 (6). Then, the third application process can use the object (7).

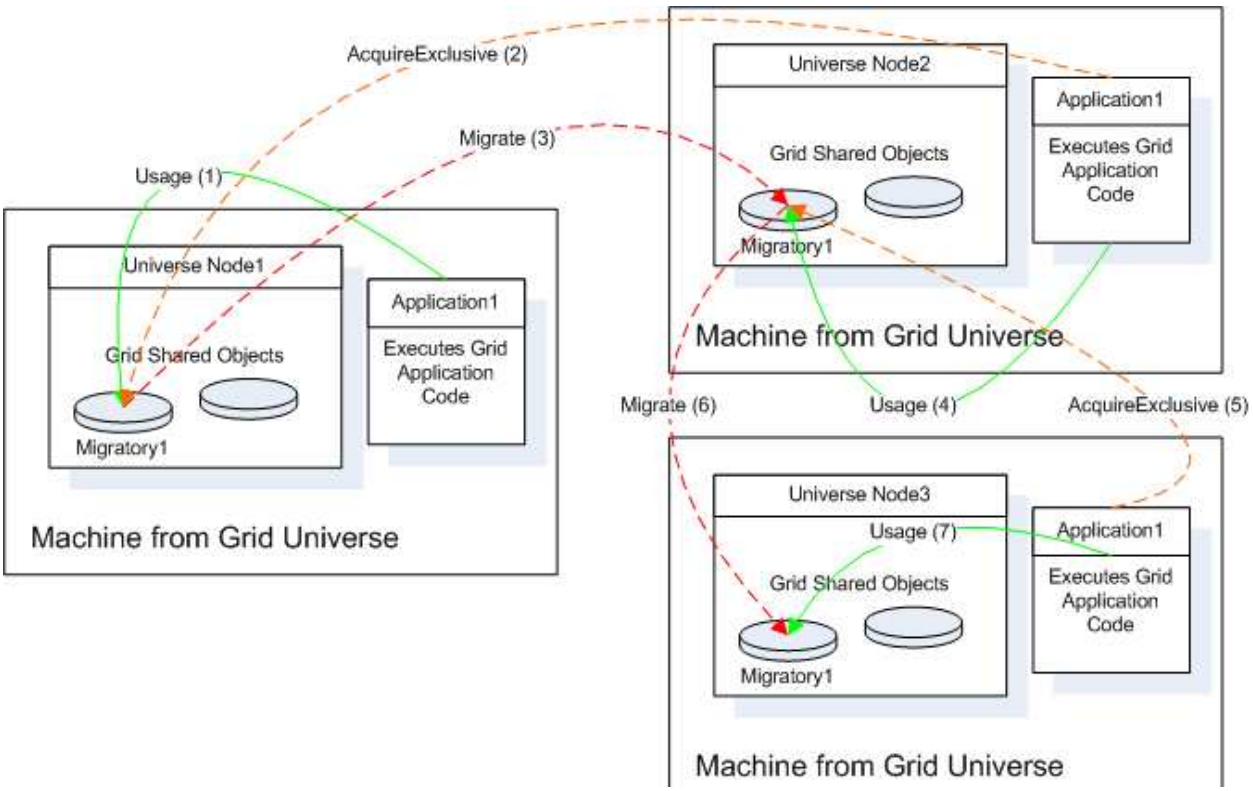

Figure 35: Migratory Objects Handling

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

Migratory objects require a machine specific locking mechanism which can be handled in the arbitration methods on each node, without making use of the token based arbitration between primary nodes. The mechanism is similar to the private objects, and relies on a local read-write mutex too. Each universe node that contains a migratory grid object has a mutex associated with each object type instance. In order to ensure reference location transparency and overcome the effects of the second consequence, each node performs a lookup on the supplied grid object reference in order to identify the correct node where the object has eventually migrated to.

```
// For evey node Ni
RWMutex[]       – Read/Write mutex that protects a resource for read/write access according to
                entry consistency semantics, one per each object type
self            - the identity of the current node

/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut)
        if (Type(ref) == MIGRATORY)
                GridObjectRef newRef = LookupRef(ref)
                if (ref.DataNode != newRef.DataNode)
                        // Object has been migrated, forward request to new node
```

```
                        return Call Acquire(newRef, timeOut) on ref.DataNode
                end if
                RWMutex[ref.OID].ReadLock()
        end if
        // Code continues from the generic grid object handling
        . . .

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref)
        if (Type(ref) == MIGRATORY)
                RWMutex[ref.OID].ReleaseLock()
        end if
        // Code continues from the generic grid object handling
        . . .

/** Looks up the correct node where the data referred by the reference has been migrated.
        @param  ref             Grid object reference.
        @return                 The correct reference.

*/
LookupRef(GridObjectRef ref)
        // identify the referred grid object
        GridObject o = ref.GetObject()
        if (o = null)
                // object has been migrated
                return Call FindGridObject(ref.OID, ref.ClientNode) on ref.ClientNode.PN
        end if
        return ref

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut)
        if (Type(ref) == MIGRATORY)
                GridObjectRef newRef = LookupRef(ref)
                if (ref.DataNode != newRef.DataNode)
                        // Object has been migrated, forward request to new node
                        return Call AcquireExclusive(newRef, timeOut) on ref.DataNode
                end if
                // Object resides on this node
                RWMutex[ref.OID].WriteLock()
                // The requesting node must be different to the node where the object resides
                if (ref.ClientNode != self)
                        // identify the referred grid object
                        GridObject o = ref.GetObject()
                        // Migrate the referred object from this node to the requestor node
                        ok = Call CreateGridObject(o, ref.ClientNode) on ref.ClientNode.PN
                        if (ok = true)
                                // unregister the object from its current location
                                Call PN_RemoveGridObject(ref.OID) on PN
                                capacity = capacity – size(o)
                                delete o
                                // update reference's data node as the new node
                                // remote assignment to the process space where the ref belongs to
                                ref.DataNode = ref.ClientNode
                                // obtain the mutex on the new node where the object was migrated
                                // all references will be pointed to the new node from now on
                                Call AcquireExclusive(ref, timeOut) on ref.ClientNode
                                // operations on this node are complete as the object is migrated
                                RWMutex[ref.OID].ReleaseLock()
                        end if
                end if
        end if
        // Code continues from the generic grid object handling
```

Consequences. The following consequences apply for grid migratory objects:

1. A migratory object is migrated only if there is space available on the node residing on the same machine with the caller client node.
2. After an object has been migrated to another node, all previously obtained references are invalid.
3. Whenever an invalid reference is used for the first time, the reference must be resolved.

5.4 Producer-Consumer Objects Handling

Producer-Consumer objects defined in *Definition 3.15* are objects that are written (produced) by a single process and read (consumed) by multiple other processes. The number of consumer processes can be arbitrary and new processes can become consumers at any time. The frequency of write operations directly determines the state update mechanism: write-update or write-invalidate. If the write frequency is low and read frequency is high, it would be advantageous to perform an eager write update protocol at release time. If the situation is opposite to this one, and the write frequency is high, an eager update protocol (e.g. perform replica update immediately at release time) would only waste bandwidth as changes might not reach the destination until a new change is propagated. The same applies in case of lightweight or heavyweight objects. Such a decision cannot be made at the system design time, but it represents an application characteristic that need to be address at deployment time. The default synchronization mechanism for generic objects is write-invalidate. We consider that at deployment time, the user can specify the rules to be followed for producer-consumer update: update or invalidate, based on the same primitives defined in the replication case. The overall characteristics of producer-consumer objects are summarized in Table 7. A visual representation of producer-consumer objects handling is shown in Figure 36.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Producer-Consumer	n	yes	no	n	1

Table 7: Producer-Consumer Object Characteristics

In Figure 36 we depict the scenario where a producer-consumer object “Producer-Consumer1” is replicated to two other nodes and is written (produced) by a process co-located with Universe Node1, and read (consumed) by two processes co-located with Universe Node2 and Universe Node3. It is important to note that the object is written only in Universe Node1 and only read on other nodes.

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

The generic algorithm requires adaptations only if the synchronization policy is “write-update” and an eager synchronization protocol is used. In the generic solution, a list of nodes that have acquired the object in non-exclusive mode is stored in the token structure. The list is dynamically updated so that every time a release is triggered the corresponding node is removed and added when an acquire is invoked. In case of producer-consumer objects, a second list is used that holds all nodes that are identified as consumer nodes, which are nodes that are only acquiring in non-exclusive mode. Upon release in exclusive mode as a result of a write operation, the updated version of the shared data is synchronized to all known consumer nodes. For this reason, we augment the token structure by adding a list of consumer

nodes, ***consumers***. The following pseudo-code reflects these changes, which are marked in bold style that affects all primary nodes.

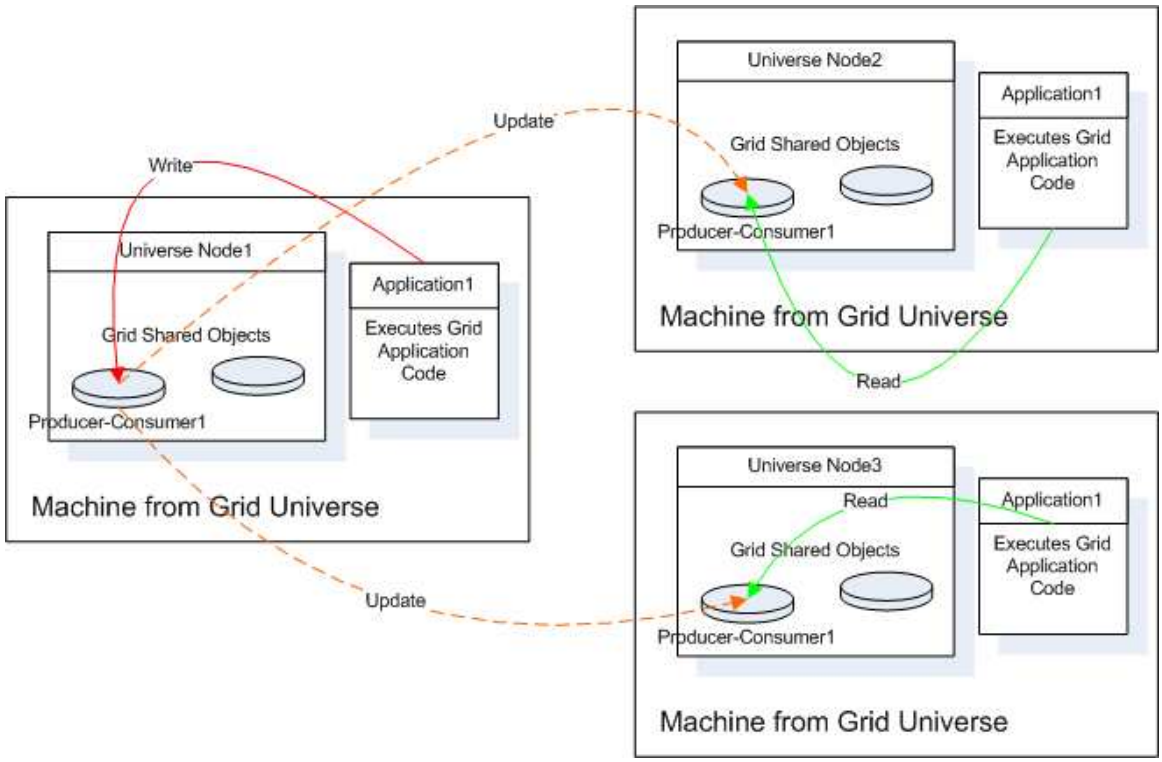

Figure 36: Producer-Consumer Objects Handling

```
// For evey primary node PNi
Struct{
        Boolean : exclusive    - specifies if the token is owned exclusively
        Node    :  nonex[]     - nodes that hold the token in non-exclusive mode
        Node    :  latest[]    - nodes that hold the most recent copy of an object with OID
        Node    :  consumers[] – consumer nodes of Producer-Consumer type

}Token : token[]       - array of tokens for each object identifier

/** Performs an acquire request.
        @param  ref           Grid object reference.
        @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_Acquire(GridObjectRef ref, Node callerNode):
        if owner[ref.OID] != 0 then
                // the node does not have the token
                // node registers to the token owner as non-exclusive access
                Send Request_Acquire(ref, callerNode, self) to owner[ref.OID]
                Wait for Acquire_Granted(ref.OID, latestNodes)
        else
                // this primary node has the token
                Wait until (token[ref.OID].exclusive == false)
                // now the token can be held non-exclusively
                Add callerNode to token[ref.OID].nonex
                latestNodes = token[ref.OID].latest
                if (ref.DataNode not in latestNodes) then
                        Add ref.DataNode to token[ref.OID].latest
                end if

                // update consumer nodes if not already known
                if (Type(ref) == PC and callerNode not in token[ref.OID].consumers)
```

```
                        Add callerNode to token[ref.OID].consumers
                end if
        end if
        // Code continues from the generic grid object handling
        . . .

/** Processes an acquire request.
        @param  ref             Grid object reference.
        @param  requestorNode   The caller node from where the operations has been requested from.
        @param  callerPN        Primary node from where the request has been issued.
*/
Receive_Request_Acquire(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if (owner[ref.OID] == 0) then
                // this node has the token
                Wait until (token[ref.OID].exclusive == false)
                Add requestorNode to token[ref.OID].nonex
                // send the token granted to the requesting primary node
                Send Acquire_Granted(ref.OID, token[ref.OID].latest) to callerPN
                Add ref.dataNode to token[ref.OID].latest
                // update consumer nodes if not already known
                if (Type(ref) == PC) and requestorNode not in token[ref.OID].consumers)
                        Add requestorNode to token[ref.OID].consumers
                end if
        else
                // non-root node, forward request
                Send Request_Acquire(ref, requestorNode) to owner[ref.OID]
        end if

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref, Node callerNode):
        requesting[ref.OID] = false
        if (token[ref.OID].exclusive == true) then
                // synchronize all “consumer nodes”
                if (Type(ref) == PC)
                        Call SynchronizeAll(ref, token[ref.OID].consumers) on ref.DataNode
                end if
                // Code continues from the generic grid object handling
        else
                // Code continues from the generic grid object handling
        end if
```

5.5 Read-Mostly Objects Handling

Read-Mostly objects introduced in *Definition 3.16* represent a more general case of a producer-consumer object, where the number of writers is not limited to one anymore. If write operations (namely *AcquireExclusive*) occur on the same universe node, the situation is similar to the producer-consumer case, but if the write operations are issued from different nodes and the write frequency is much lower than read frequency, we identify those objects as read-mostly objects. The major difference compared to producer-consumer objects is that multiple writers from different nodes require a token exchange between universes. This situation does not occur in the case of producer-consumer objects, as the number of writers is limited to one. Similar to producer-consumer objects, we adopt a configurable synchronization mechanism. The overall characteristics of read-mostly objects are summarized in Table 8. A visual representation of read-mostly objects handling is presented in Figure 37.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Read-Mostly	n	yes	no	n	<<n

Table 8: Read-Mostly Object Characteristics

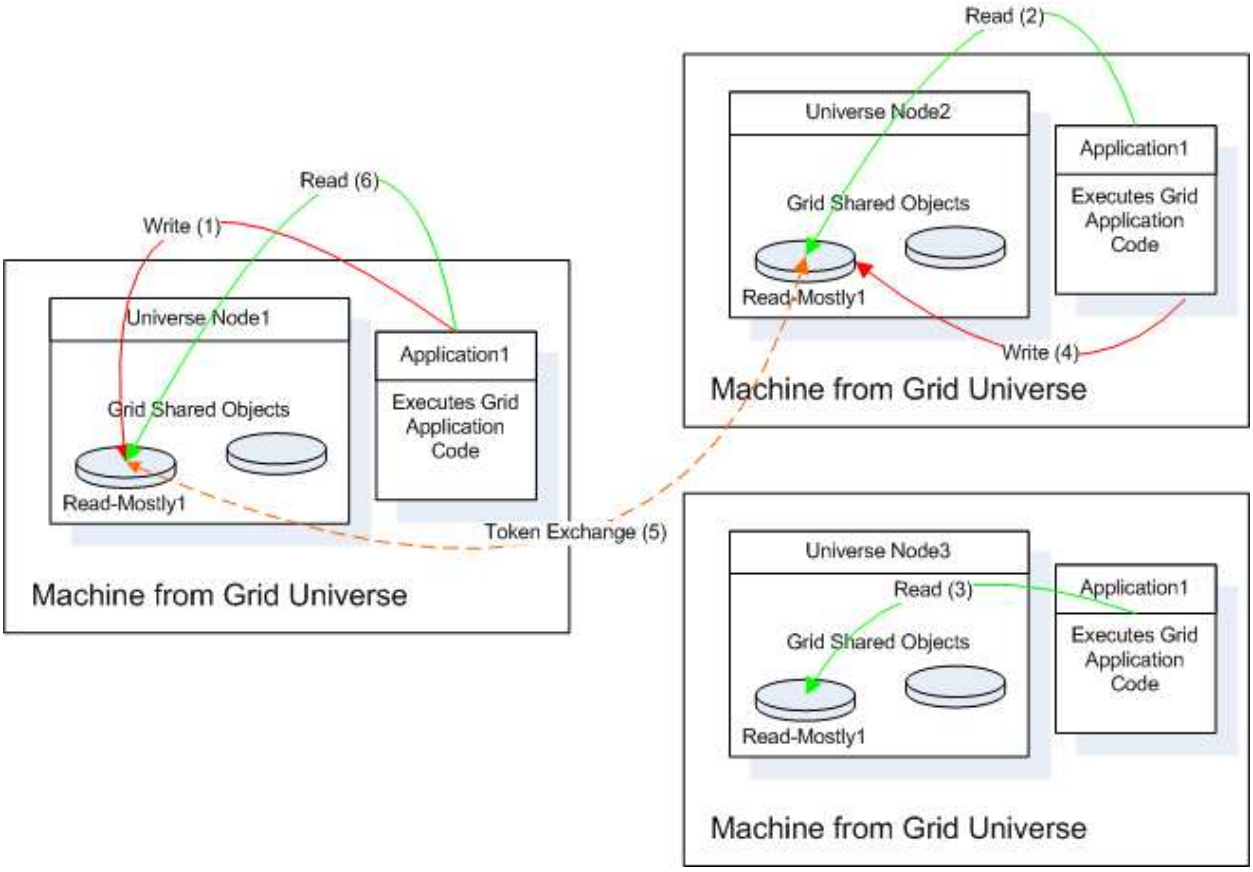

Figure 37: Read-Mostly Objects Handling

Figure 37 considers a situation where a read-mostly object is replicated to three different universe nodes and an application is running one process on each of the three nodes. The first process is acquiring the read-mostly object exclusively (1) and performs a series of write operations. Next, the second process is acquiring the same object, waiting of course until the first process releases the object. The token is not exchanged to the second node, but only the object's data is synchronized. A similar action is triggered by the third application process (3). Next, the second process wants to write the object and acquires the object exclusively (4). As a result, the token is transferred to the second universe node (5). Next, the object replica from universe two is written by the second application process. Finally, the first application reads the content of its replica by acquiring the object in non-exclusively (6).

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

Read-mostly objects handling requires similar adaptations as producer-consumer objects only if the synchronization policy is "write-update" and an eager synchronization protocol is used. The difference to producer-consumer objects is that the writers are also part of the "reader" list, so that when eager synchronization is performed, the writer nodes are also updated. The following pseudo-code reflects these changes, which are marked in bold style that affects all primary nodes.

```
// For evey primary node PNi
Struct{
        Boolean : exclusive    - specifies if the token is owned exclusively
        Node    :  nonex[]     - nodes that hold the token in non-exclusive mode
        Node    :  latest[]    - nodes that hold the most recent copy of an object with OID
        Node    :  consumers[] – consumer nodes of Producer-Consumer type

}Token : token[]       - array of tokens for each object identifier

/** Performs an acquire request.
        @param  ref           Grid object reference.
        @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_Acquire(GridObjectRef ref, Node callerNode):
        // Same code as in the Producer-Consumer case
        if owner[ref.OID] != 0 then
                . . .
        else
                . . .
                // update consumer nodes if not already known
                if (Type(ref) == RM and callerNode not in token[ref.OID].consumers)
                        Add callerNode to token[ref.OID].consumers
                end if
        end if
        // Code continues from the generic grid object handling
        . . .

/** Performs an acquire exclusive request.
        @param  ref           Grid object reference.
        @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_AcquireExclusive(GridObjectRef ref, Node callerNode):
        requesting[ref.OID] = true
        if owner[ref.OID] != 0 then
                // Code continues from the generic grid object handling
                . . .
        else
                // Code continues from the generic grid object handling
                . . .
        end if
        // update read-mostly nodes if not already known
        if (Type(ref) == RM and callerNode not in token[ref.OID].consumers)
                Add callerNode to token[ref.OID].consumers
        end if
        // Code continues from the generic grid object handling
        . . .

/** Processes an acquire request.
        @param  ref           Grid object reference.
        @param  requestorNode  The caller node from where the operations has been requested from.
        @param  callerPN      Primary node from where the request has been issued.
*/
Receive_Request_Acquire(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if (owner[ref.OID] == 0) then
                // Code continues from the generic grid object handling
                . . .
                // update consumer nodes if not already known
                if ((Type(ref) == RM) and requestorNode not in token[ref.OID].consumers)
                        Add requestorNode to token[ref.OID].consumers
                end if
        else
                // non-root node, forward request
                Send Request_Acquire(ref, requestorNode) to owner[ref.OID]
        end if

/** Processes an acquire exclusive request.
        @param  ref           Grid object reference.
        @param  requestorNode  The caller node from where the operations has been requested from.
        @param  callerPN      Primary node from where the request has been issued.
*/
```

```
Receive_Request_AcquireExclusive(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if owner[ref.OID] = 0 then
                // update read-mostly nodes if not already known
                if (Type(ref) == RM and requestorNode not in token[ref.OID].consumers)
                        Add requestorNode to token[ref.OID].consumers
                end if
                // Code continues from the generic grid object handling
        end if
        // Code continues from the generic grid object handling

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref, Node callerNode):
        requesting[ref.OID] = false
        if (token[ref.OID].exclusive == true) then
                // synchronize all "read-mostly nodes"
                if (Type(ref) == RM)
                        Call SynchronizeAll(ref, token[ref.OID].consumers) on ref.DataNode
                end if
                // Code continues from the generic grid object handling
        else
                // Code continues from the generic grid object handling
        end if
```

5.6 Result Objects Handling

Definition 3.17 describes result objects as grid shared objects that consist of several non-conflicting parts that can be updated simultaneously. As a consequence, whenever a result object's part is written, no mutual exclusion is required. However, the interface specification requires that the *AcquireExclusive* method to be called, in order to signal that an object's part is to be modified. The object's overall state is composed on the node where a normal acquire has been invoked. The replication rate of such object is medium so that a good tradeoff between data locality and data synchronization penalty is reached. Object replicas allow local writes that need not travel across large latency connections. Of course, during object synchronization, some parts of the object's content need to travel across large latency connections in order to obtain the complete object state. The overall characteristics of read-mostly objects are summarized in Table 9. A visual representation of result objects handling is shown in Figure 38.

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Result	n	yes	no	1	n

Table 9: Result Object Characteristics

In the depicted scenario, each application process is writing simultaneously one part of the result object: the process from universe node1 is writing the third part, the process from universe node2 is writing the first part and the one from universe node3 is writing the second one. At some point in time the application process co-located with the universe node2 is issuing an acquire request, signalling that it requires the entire object state. Next the object state is composed on the universe node2 out of the received parts from other universe nodes.

Algorithm adaptations. In the following code snippet, we present the required actions we consider to execute in order to fulfill the semantics of this object type. For the sake of simplicity and conciseness, we only present the changes in respect to the generic grid object approach. The complete system solution and logic are presented in Appendix A.3.

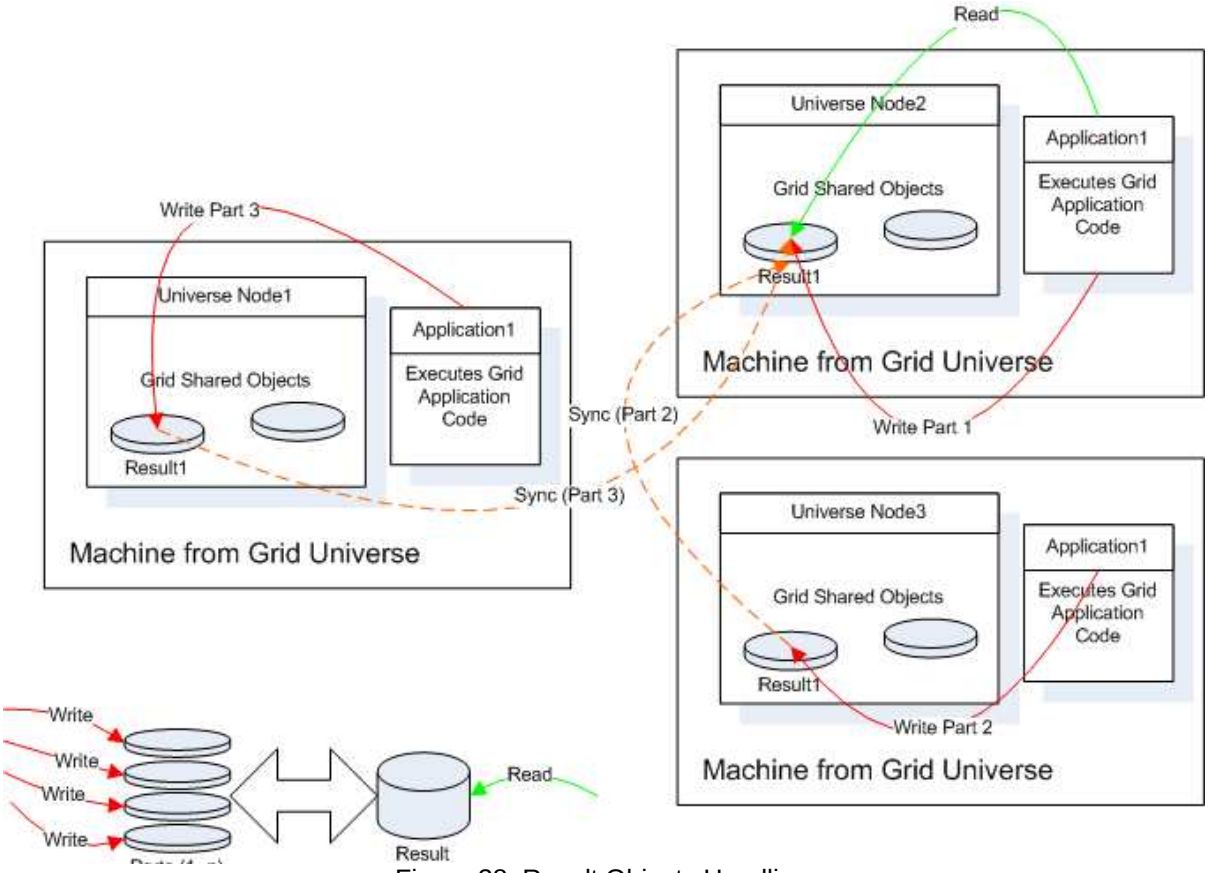

Figure 38: Result Objects Handling

Result objects require a slightly different handling than the generic grid objects. From the usage perspective, AcquireExclusive is called when an object's part is modified. When the object's state is read, Acquire is called, but all other write operations must be completed at this point. This means that AcquireExclusive can be invoked simultaneously, but Acquire can be invoked only by one caller at a time. In other words, the semantics of Acquire and AcquireExclusive operations is the opposite of the generic object type. Basically there are two aspects to be taken care of: object part versioning and maintaining a correct writer set. Object versioning is maintained by each node. Whenever an object part is written for the first time, the original version is stored. We consider as original version either the original version of that part (or after an object replication) or the version obtained after the object's state has been synchronized. The second change refers to the writer set. Whenever an object is acquired in exclusive mode, the token is not required anymore as there is no arbitration, but the node's identity is maintained in a list of writers, that we refer as the writer set. When an normal acquire is called, the node where the result object's state need to be assembled and synchronized receives all updated parts, based on the writer set's versioning information.

```
// For every primary node PNi
/** Performs an acquire exclusive request.
       @param  ref           Grid object reference.
       @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_AcquireExclusive(GridObjectRef ref, Node callerNode):
       if (Type(ref) == RES)
```

```
                if owner[ref.OID] != 0 then
                        // the node does not have the token
                        // node registers to the token owner as non-exclusive access
                        Send Request_Acquire(ref, callerNode) to owner[ref.OID]
                        Wait for Acquire_Granted(ref.OID, latestNodes)
                else
                        // this primary node has the token
                        Wait until (token[ref.OID].exclusive == false)
                        // now the token can be held non-exclusively
                        Add callerNode to token[ref.OID].nonex
                end if
                Send AcquiredExclusive to callerNode
        end if
        // Code continues from the generic grid object handling
        . . .

/** Performs an acquire request.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
*/
PN_Acquire(GridObjectRef ref, Node callerNode):
        if (Type(ref) == RES)
                requesting[ref.OID] = true
                if owner[ref.OID] != 0 then
                        // the node does not have the token
                        // token must be requested
                        Send Request_AcquireExclusive(ref, callerNode) to owner[ref.OID]
                        owner[ref.OID] = 0
                        Wait for Token(ref.OID, latestNodes)
                else
                        // the node has the token
                        Wait until (token[ref.OID].nonex is empty)
                        Wait until (token[ref.OID].exclusive == false)
                end if
                // perform object state synchronization
                Call AssembleObject(ref, token[ref.OID].nonex) on ref.DataNode
                // reset token
                token[ref.OID].latest = ref.DataNode
                token[ref.OID].nonex = 0
                token[ref.OID].exclusive = true
                Send Acquired to callerNode
        end if
        // Code continues from the generic grid object handling
        . . .

/** Assembles the object state residing on this node out of the updated values of replicas
residing in a list of nodes.
        @param  ref             Grid object reference.
        @param  nodes           Array of nodes that hold updated versions of object parts.
*/
AssembleObject(GridObjectRef ref, Node[] nodes):
        // Construct the parts of the object
        GridObject o = ref.GetObject()
        Part parts[] = o.GetParts()
        Part changedPart

        index = 0
        for each part in parts
                index++
                for each n in nodes
                        // part is the local part that might have been written somewhere
                        // changedPart is the out parameter, the most recent written part
                        changed = (Call GetObjectPart(index, part, changedPart) on n //remote call
                        if (changed = true)
                                parts[index] = changedPart
                                // first found is the correct non-conflicting part
                                break
                        end if
                end for
        end for
```

Consequences. The following consequences apply for grid result objects:

1. A result object's part represents an indivisible data unit that is typically updated via a single write operation.
2. Each time a result object's part is modified, the original version is stored by the node so that it is known if a part has been changed or not.
3. An object can be reconstructed out of its parts.
4. Violations of the interface and object semantics result in undefined object state upon read synchronization (e.g. if a part is written by more than one client).

5.7 Write-Mostly Objects Handling

According to *Definition 3.18,* write-mostly objects as a step more general than result objects by removing the constraint that an object can be decomposed in non-conflicting parts. In addition the number of readers is not limited to one anymore. Write-mostly objects are generic grid shared objects that are written (acquired in exclusive mode) very frequently and read (acquired in normal mode) seldom. Such objects do not benefit of a high replication ratio, but rely on low replication rates and lazy synchronization mechanisms. The overall characteristics of write-mostly objects are summarized in Table 10. A visual representation of write-mostly objects handling is shown in Figure 39, where writers are located in all universe nodes and the only reader is located in universe node number three.

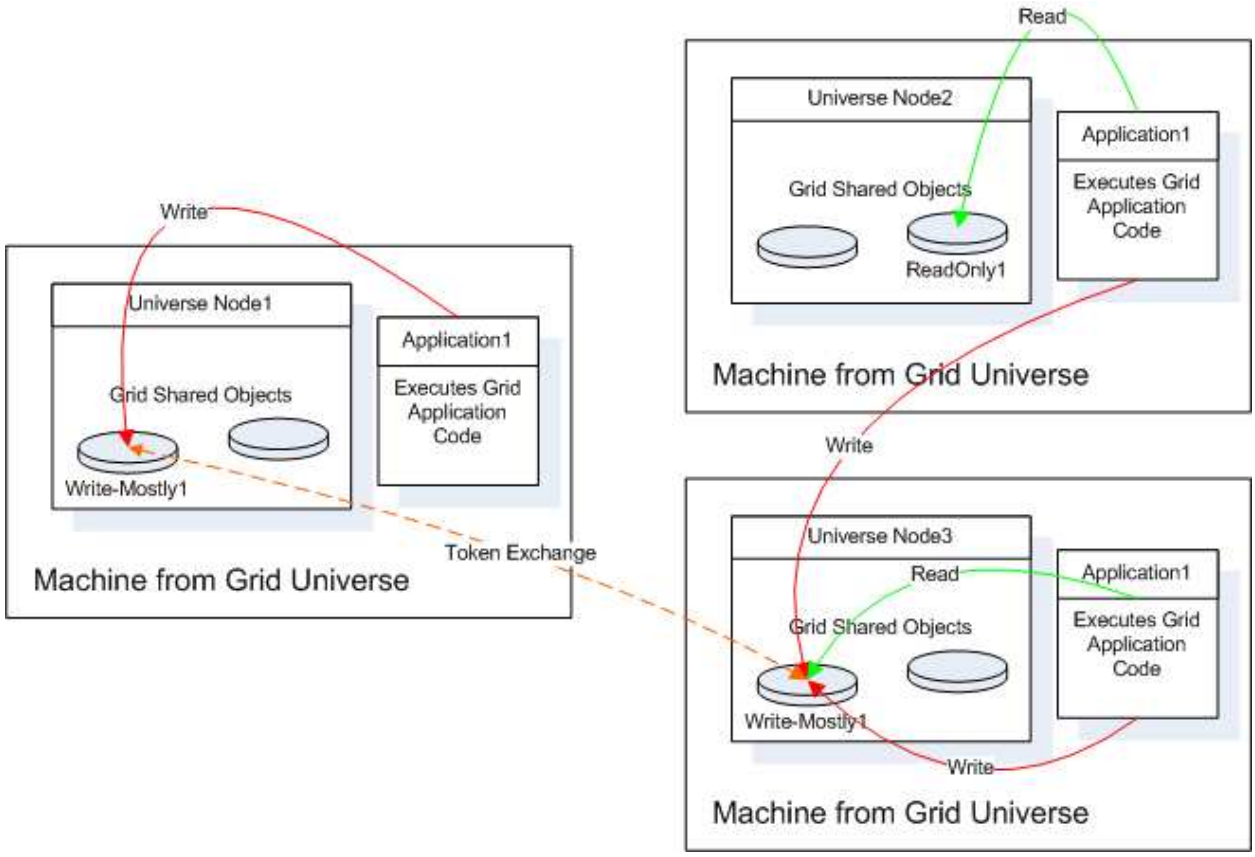

Figure 39: Write-Mostly Objects Handling

Grid Object Type	Number of Objects	Replicate	Migrate	Readers	Writers
Write Mostly	n	yes	no	<<n	n

Table 10: Write-Mostly Object Characteristics

Algorithm adaptations. Write-mostly objects handling do not require any particular algorithm adaptation. The generic object handling which relies on lazy synchronization suites very well this kind of object handling. Write-mostly objects carry semantic information used to properly choose the replication rules.

5.8 Object Type Transformations

During an application execution one can start operating on some type of objects that are initially created. However, due to special conditions in the application logic, the original object type might not be suitable anymore and a new object type could be required instead of the original one. One simple example is during a computation where a final or intermediate result could be supplied as a read-only object. Such an object can be highly replicated and could be easier to reach for processes located "at the edges" of the grid universe. Similarly, as a computation evolves, some shared object could be used only locally from a certain point in time, thus private objects could be used instead with a much lower synchronization penalty. In order to support grid shared object type transformation, we supply an extension to the original interface of the grid object and its corresponding reference, as shown in Figure 40.

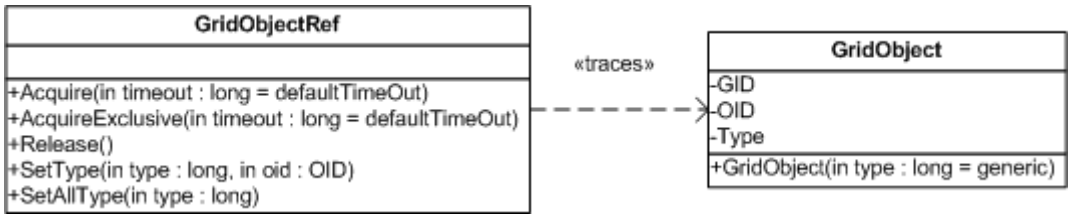

Figure 40: Grid Object Reference Type Handling

The semantics of both SetType methods are described next:

GridObjectRef::SetType(long type, OID oid) – changes the type of the object referred by the reference to *type* and associates new user provided identifier, oid.

GridObjectRef::SetAllType(long type) – changes the type of the object referred by the reference and all its replicas to *type*.

Discussion. The first method changes the type of the directly referred object. As replicas of the designated object might exist, the caller must provide a new and unique identifier for the updated type object. If the provided identifier is not unique, undefined behavior shall be expected as there is no guarantee which of the existing object types is returned as a result of a find operation. The operation can be called at any time the object referred by the reference.

```
// For every node N
/** Changes the object type of an object.
       @param  ref    Grid object reference.
       @param  type   The new type of the object instance.
       @param  oid    The new identifier of the object.
*/
SetType(GridObjectRef ref, long type, OID oid):
```

```
        // the node does not have the token
        if (Call AcquireExclusive(ref, self) on self) = true)
                // the refered object resides on this node
                GridObject o = ref.GetObject()
                o.SetType(type)
                o.SetOID(oid)
                // release the object
                Call Release(ref, self) on self
        end if
```

The second method changes the type for all objects identified by the same user provided identifier. In this case all objects keep their identifiers, but depending on object's type, synchronization might be performed or not. Table 11 summarizes the synchronization requirements. For example, a change type to a read-only, private, producer-consumer of migratory object from any other object type requires that all newly transformed objects be synchronized upon transformation. In case of migratory object transformation, if more than one object is transformed, it requires to dispose all but one such object and keep the latest version. Result, write-mostly and the generic object type transformations do not require any particular synchronization operations.

Transition From	Transition To	Synchronization
Any	Read-only	All objects
Any	Private	All objects
Any	Migratory	Latest and dispose all (n-1) copies
Any	Producer-Consumer	All objects
Any	Read-Mostly	All objects
Any	Result	None
Any	Write-Mostly	None
Any	Generic	None

Table 11: Object Transformation Synchronization

```
/** Changes the object type of an object and all its replicas
        @param  ref    Grid object reference.
        @param  type   The new type of the object instance.
*/
SetAllType(GridObjectRef ref, long type):
        // the node does not have the token
        if (Call AcquireExclusive(ref, self) on self) = true)
                // update all objects in all universes
                Call UpdateObjectType(ref, type) on PN
                // release the object
                Call Release(ref, self) on self
        end if
```

5.9 Object Transfer Protocol

One of the major design guidelines has been to separate deployment configuration from system built-in logic. As a result we have split the logic for object replication into one part supplied to the system through deployment information and the generic implementation part which provides the logic and the realization for the replication rules and primitives. In this way, one can tailor different system behaviors as required by the deployed application. In a similar fashion, we define object synchronization protocol as part of the deployment information. By default, the generic grid object relies on lazy synchronization. However, we have shown that read-mostly objects could benefit out of eager synchronization protocols such as write-update. As a consequence, the choice of the synchronization protocol for read-mostly objects and producer-consumer is to be supplied via deployment configurations.

In the previous chapter, when referring to object synchronization, we did not refer to any specific protocol, but we implied that a coarse grained synchronization is performed and the complete object data is copied from one node to another. This protocol works well for small objects, but as it relies heavily on the ability to represent an object as a collection of bytes (serialization), it is not feasible for large or very large objects. We have addressed indirectly this problem in case of result objects, which rely on an explicit structure by breaking each result object is indivisible units referred as parts. This approach does not address again the case of large or very large parts. In case of large and very large objects, we adopt a different technique based on binary differences. The concept is well known in operating systems such as Unix where files need to be patched using the smallest possible patch image as fast as possible. Instead of sending the complete image, a binary difference is generated and sent instead of the complete object. The patch is applied to the original content and the final object is obtained. Using for example the ***bsdiff*** algorithm, one could reduce the transported data from 16MB to 300KB, for a 16MB object. We consider again the object synchronization protocol as part of the deployment information, giving the system architect the freedom to choose different object synchronization protocols for different object types.

5.10 Putting All Together

The complete pseudo-code solution that describes the algorithm for the presented system model is listed in Annex A.3. The pseudo-code illustrates the operations performed on each universe node and primary node and covers both generic objects as well as all specialized objects.

6 Experiments and Theoretical Analysis

Distributed systems' analysis is a very important, complex and sensitive topic. One of the most common analysis domains relate to performance analysis where performance related aspects of a certain system are aimed to be highlighted. In many situations these aspects refer to response time for various operations. A simple example is the total execution time of a distributed algorithm, given certain input data and a particular system deployment. Other aspects of distributed systems' analysis relate to the number and capacity of used resources in order to complete a certain task. Quality related analysis such as different kinds of statistical information represents another dimension of distributed system's analysis (e.g. quality of service). Independent of the specific analysis that is aimed, a common problem in distributed systems and especially in grids is to be able to reproduce a given system state such as the number of deployed machines, characteristics of the connectivity layer (e.g. bandwidth, latency, network congestion) and machine characteristics (system load, free memory, resource distribution). As one can immediately notice, the deployment of real-life scenarios in an open environment leads to a very high number of possible combinations. Considering that most grid systems are used by a large number of users, it is very difficult to ensure a certain global system usage at a given time. Although resource reservation systems are widely used, they do not offer hard guarantees. As a result, most system evaluations are preferred to be done in ideal conditions where for example in a grid environment only the application under test is running (fixed resource allocation scheme) and the connectivity layer is closed to the outside world. In these cases, the number of deployed machines is most of the times quite modest.

Evaluating a grid system in ideal conditions is straightforward, but it opens the question of reproducibility likelihood, meaning that if one wants to reproduce a given experiment, one must ensure a similar environment. Due to the complexity of grid systems, sometimes this requirement cannot be achieved. Worse, a real-life experimental scenario is almost impossible to reproduce in case of a large scale distributed application deployed on a wide area grid. This brings us to the idea of considering other means of system evaluation that could give the possibility to correlate results from different experiments. We aim to perform system analysis on three different directions: theoretical analysis, prototype-based analysis and computer aided analysis. We distinguish three main analysis domains while analyzing a grid system: performance analysis, resource related analysis and quality related analysis.

Performance related analysis refers mostly to the elapsed time for a given operation such as response time for a given request or global completion time. ***Resource related analysis*** refers mostly to the resource usage in order to complete a given operation such as CPU or memory usage. It might also refer to resource status information like latency and bandwidth. ***Quality related analysis*** comprises statistical information that is collected during the execution time of an application in a distributed system. It may refer to operation acceptance/revocation ratio, time-based availability, throughput, correctness level etc. Generally speaking, quality related analysis is closely related to particularities of the analyzed system as it aims to highlight domain specific aspects opposite to generic aspects found in the other two categories.

6.1 Evaluation Criteria

In order to analyze the abstract model introduced in [128], we introduce several evaluation criteria as analysis metrics that shall be used while referring to different aspects of our system. We associate an abbreviation to each of the criteria in order to refer to them easily. We consider that a test application is running in the grid system which injects different stimuli into the system in order to analyze certain aspects.

6.1.1 Performance Criteria

In the context of performance analysis, we define the following measurements which serve as a performance evaluation criteria for our model:

- RT – Response time [ms] – represents the execution time for a given operation such as creating, locating or removing an object.
- CT – Completion time [ms] – represents the execution time for a distributed application, from the time the first application process starts execution until the last process finishes.
- TT – Token obtaining time [ms] – represents the time elapsed from the moment a primary node issues a token request until the token is received by the primary node. The average (AVG_TT), minimum (MIN_TT) and maximum (MAX_TT) values are considered.
- OST – Object synchronization time [ms] – represents the time required to synchronize two grid shared objects (e.g. copy the state of one object into the other object). The average (AVG_OST), minimum (MIN_OST) and maximum (MAX_OST) values are considered.
- AQT – Acquire time [ms] – represents acquire time, which is the time elapsed from the moment of issuing an acquire request from a node until the acquire operation is granted on the referred data. The average (AVG_AQT), minimum (MIN_AQT) and maximum (MAX_AQT) values are considered.
- AQET – Acquire exclusive time [ms] – represents acquire exclusive time, which is the time elapsed from the moment of issuing acquire exclusive request until the operation is granted on the referred data. The average (AVG_AQET), minimum (MIN_AQET) and maximum (MAX_AQET) values are considered.

6.1.2 Resource Criteria

In terms of resource related analysis, we define the following measurements which serve as an evaluation criteria for our model:

- MEM – MEM usage [MB] – represents the memory demand per each node.
- OP_SCOPE – Operation scope [nodes] – represents the number of nodes involved in the execution of one operation (e.g. searching scope).

6.1.3 Quality Criteria

In terms of quality related analysis, we define the following measurements which serve as an evaluation criteria for our model:

- AQSR – Acquire success rate [%] – represents the average success rate for all issued acquire operations during the application execution. An acquire operation is successful if the operation is granted within the demanded timeout. AQSR can relate to either a node, universe or to the grid universe.
- AQESR – Acquire exclusive success rate [%] – represents the average success rate of acquire exclusive operations for all issued acquire exclusive operations during the application execution. An acquire exclusive operation is successful if the operation is granted within the demanded timeout. AQESR can relate to either a node, universe or to the grid universe.
- TRAQ – Throughput for acquire [operations/s] – represents the number of successful acquire requests per second that are issued in the system.
- TRAQE – Throughput for acquire exclusive [operations/s] – represents the number of successful acquire exclusive requests per second that are issued in the system.

6.2 Experiments

This section describes different experiments that aim to highlight performance, resource as well as qualitative aspects of the grid universe model. Evaluation of a generic application is most of the times a very hard task due to the lack of information about the application's external conditions. On the other hand, a particular application (e.g. traveling salesman problem) might not provide sufficient information so that a generic profile can be synthesized. In other words it is hard to devise a general behavior rule based on a particular set of application specific interactions (both data and service). As a result, opposite to evaluating a concrete application, we aim to evaluate different interaction patterns that can be interactions within a real-life application. Any application can be decomposed into a set of such interactions patterns.

In the following, it is considered to have a number of *m* universes deployed. Without losing any generality, each of the *m* universes contains a number of *n* nodes. Of course imbalanced universes with different number of nodes can be considered as well, but it complicates the analysis very much. In addition, as it will be pointed out later, an imbalanced node distribution does not change the observation results. Each of the *n* nodes has a capacity *c*, thus there is a homogeneous node distribution across universes. Depending on the experiment, at a given time a number of *p* processes are running in the grid universe where $p <= m \times n$. An experiment defines a concrete interaction pattern which focuses on one or more performance aspects of the global system and sets the experiment frame. Experiments might have variants. An experiment variant defines the exploration in the system parameters space where different parameter configurations are used.

6.2.1 Grid Object Search

Purpose: The purpose of this experiment is to evaluate the characteristics of the search operation in the grid universe.

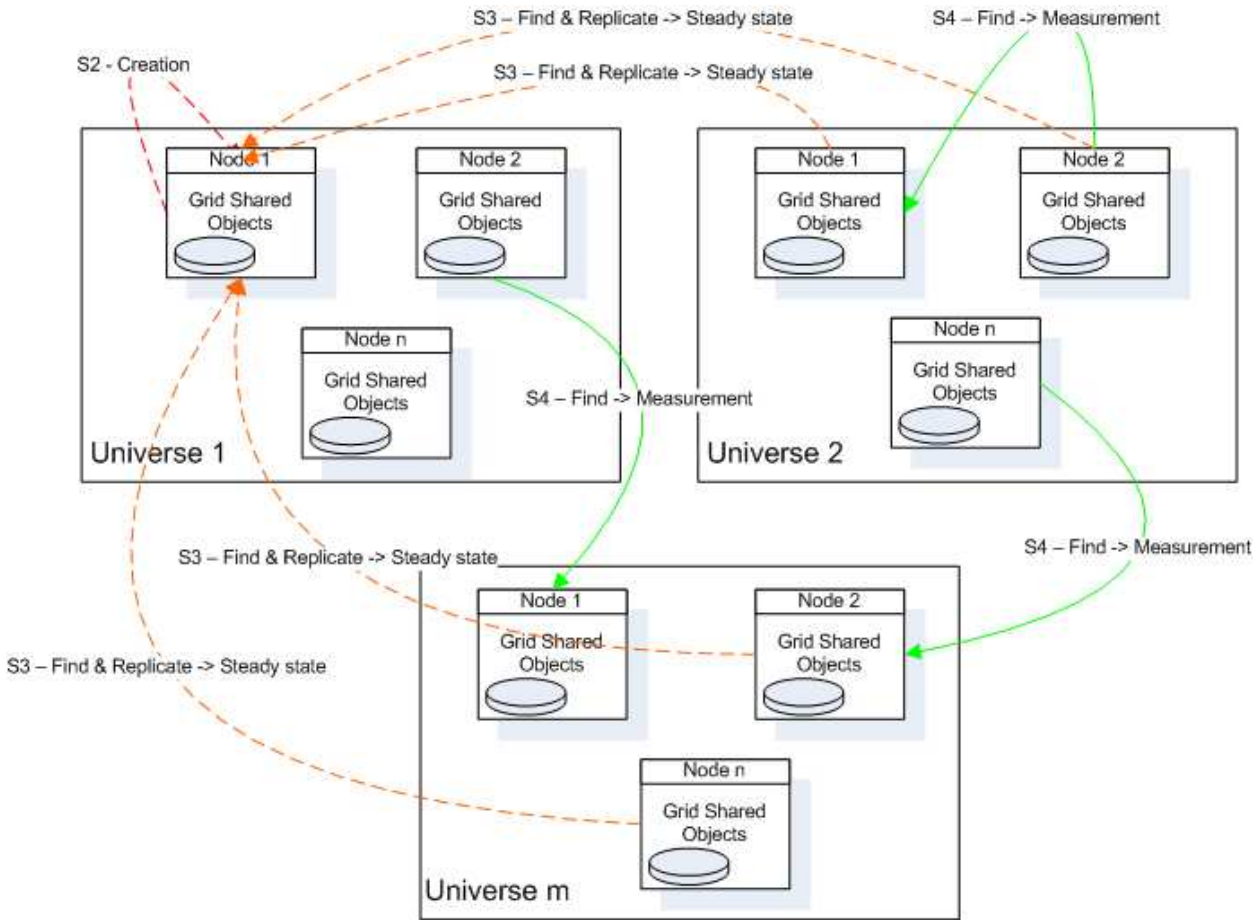

Figure 41: Grid Object Search

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one process on every grid node within the grid universe.
- [S2 - Creation] One of the p processes creates a number of o grid objects in the grid universe within a range of numerical object identifiers *[OID1, OID2].*
- [S3 - Steady state] Each process out of the p processes issues a search request in the domain of the object identifiers, in order that objects are eventually replicated and a steady state is reached.
- [S4 - Measurement] Each process performs a random search operation in the domain of the object identifiers that have been created and the evaluation criteria are logged.

Performance evaluation criteria: The following performance evaluation criteria should be measured:

- RT – response time for search operation [ms]

Experiment variants:

- [V1 – Replication] The following replication policies shall be used: “one object per universe”, “n/2 objects per universe” and “one object per node”.

- [V2 – Object Type Count] The number of different object types that are created in S2 should be {1, 10, 100, 500, 1000}.
- [V3 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.

Expected results: A decrease in search time is expected in case when object replication is enabled. Second, the search time should not be directly dependent on the number of nodes and number of objects.

Figure 41 illustrates the operations sequences defined in this experiment. The experiment's states are marked with S1 to S4. The first three states are marked with a dotted line which denotes the steps towards the steady state. The last step, the measurement is marked with a solid green arrow.

6.2.2 Acquire Correctness

Purpose: The purpose of this experiment is to observe the correctness of the acquire operation and evaluate its performance under various conditions.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe. "One object per node" replication rule shall be used in order to maximize interaction patterns between nodes.
- [S2 - Creation] One of the p processes creates a generic grid object in the grid universe.
- [S3 - Steady state] Each process out of the p processes issues a search operation in order to trigger object replication and to reach a steady system state.
- [S4 - Measurement] Each process performs a number of 100 acquire requests issued with a delay of d ms. All evaluation criteria are logged.

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- AQT – Acquire time [ms]
- AQSR – Acquire success rate [%]
- TRAQ – Throughput for acquire [operations/s]

Experiment variants:

- [V1 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.
- [V2 – Acquire Operations] The delay between subsequent operations shall be {3000ms, 2000ms, 1000ms, 500ms, 100ms}.
- [V3 – Client count] The number of client applications is limited to one (in the grid universe) and the number of nodes is the maximum possible number.

Expected results: A 100% acquire success rate shall be noticed independent on the experiment's variants.

Figure 42 illustrates the operations sequences defined in this experiment. The experiment's states are marked with S1 to S4. The first three states are marked with a dotted line which denotes the steps towards the steady state which are identical to the previous experiment. The last step, the measurement step, is marked with a solid green arrow and shows the acquire accesses between nodes.

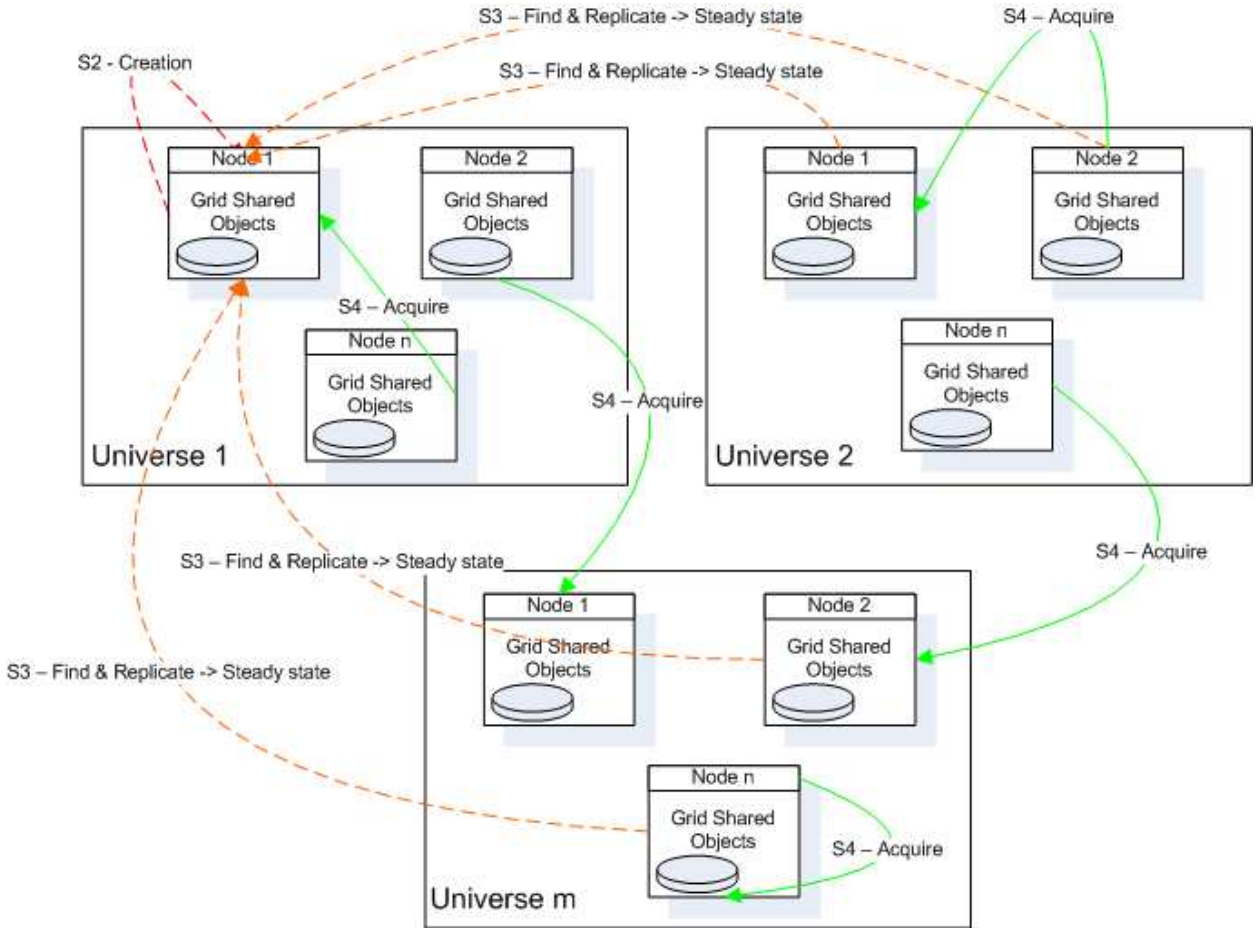

Figure 42: Acquire Correctness

6.2.3 Acquire Exclusive Correctness

Purpose: The purpose of this experiment is to observe the correctness of the acquire-exclusive operation and evaluate its performance under various conditions.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe. "One object per node" replication rule shall be used in order to maximize interaction patterns between nodes.
- [S2 - Creation] One of the p processes creates a generic grid object in the grid universe with the following data: a user provided data content (string) and a version (number) that is incremented automatically each time the content is set.

- [S3 - Steady state] Each process out of the *p* processes issues a search operation in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] Each process performs a series of 100 acquire-exclusive requests issued with a delay of *d* ms. All evaluation criteria are logged. After each acquire-exclusive operation, each process shall log the data content and the version of the object is operates on.

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for writing the remote object [ms]
- CT – Completion time for the application [ms]
- AQET – Acquire time [ms]
- TT – Token obtaining time [ms]
- AQESR – Acquire success rate [%]
- TRAEQ – Throughput for acquire exclusive [operations/s]

Experiment variants:

- [V1 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.
- [V2 – Acquire Operations] The delay between subsequent operations shall be {3000ms, 2000ms, 1000ms, 500ms, 100ms} (increasing operation frequencies).
- [V3 – Client count] The number of client applications is limited to one (per grid universe) and the number of nodes is the maximum possible number.

Expected results: In all logged grid object sequences, there should be no entries belonging to different processes that have logged the same object version (acquire exclusive correctness criteria).

6.2.4 Grid Read-Only Objects

Purpose: The purpose of this experiment is to observe the performance increase for read-only objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe. “One object per node” replication rule shall be used in order to maximize interaction patterns between nodes.
- [S2 - Creation] One of the *p* processes creates a generic grid object and a read-only grid object in the grid universe with ids *oid1* and *oid2*.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] Each process performs a series of 100 acquire requests and reading object data issued with a delay of *d* ms for both object types. All evaluation criteria are logged.

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]

- CT – Completion time for the application [ms]
- AQT – Acquire time [ms]
- AQSR – Acquire success rate [%]
- TRAQ – Throughput for acquire [operations/s]

Experiment variants:

- [V1 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.
- [V2 – Acquire Operations] The delay between subsequent operations shall be {3000ms, 2000ms, 1000ms, 500ms, 100ms}.
- [V3 – Client count] The number of client applications is limited to one (in the grid universe) and the number of nodes is the maximum possible number.

Expected results: It is expected to notice a performance increase when read-only objects are used in comparison to generic objects, as well as 100% values for quality parameters (AQSR).

6.2.5 Grid Private Objects

Purpose: The purpose of this experiment is to observe the performance increase for private objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe. No replication rule shall be used (according to private object definition).
- [S2 - Creation] One of the *p* processes creates a generic and a private grid object in the grid universe with ids *oid1* and *oid2*.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that a steady state is reached. In this case the steady state of the system is the same as the state after S2 (as no replication occurs).
- [S4 - Measurement] Each process performs a series of 100 acquire requests and reading object data, issued with a delay of *d* ms for both object types. All evaluation criteria are logged.
- [S5 - Measurement] Each process performs a series of 100 acquire-exclusive requests and writing object data, issued with a delay of *d* ms for both object types. All evaluation criteria are logged.

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- TT – Token obtaining time [ms]
- AQT, AQET – Acquire and acquire exclusive time [ms]
- AQSR, AQSR – Acquire and acquire exclusive success rate [%]
- TRAQ, TRAEQ – Throughput for acquire and acquire exclusive [operations/s]

Experiment variants:

- [V1 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.
- [V2 – Acquire Operations] The delay between subsequent operations shall be {3000ms, 2000ms, 1000ms, 500ms, 100ms}.
- [V3 – Client distribution] The client applications shall be deployed as following:
    - only on the node where the data has been created
    - only in the universe where the data has been created
    - in the entire grid universe

Expected results: It is expected to notice a medium performance increase *for requests that are issued from the same universe* and a high performance increase for *requests that are issued from the same node where the object resides*.

6.2.6 Grid Migratory Objects

Purpose: The purpose of this experiment is to evaluate the performance increase for migratory objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe. No replication rule shall be used for migratory objects (according to the migratory object definition). One object per node replication rule shall be used for generic objects.
- [S2 - Creation] One of the *p* processes creates a generic and a migratory grid object in the grid universe with ids *oid1* and *oid2*. Both objects expose a remote method *rm* that requires *d* ms to execute.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] For each type of objects, each process performs an acquire-exclusive request and a call to the object's remote method *rm* which takes *d* ms to execute. In order to mimic a migratory process and reduce token congestion, the application processes are started with a predefined delay between each other. The application processes shall be started in a sequence so that every request is issued from a different universe. The experiment is completed when all processes have held the lock once. All evaluation criteria are logged.

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- OST – Object synchronization time [ms]
- TT – Token obtaining time [ms]
- AQET – Acquire exclusive time [ms]
- AQSR – Acquire exclusive success rate [%]

- TRAEQ – Throughput for acquire exclusive [operations/s]

Experiment variants:

- [V1 – Node count] The number of nodes within universes should be {5, 10, 20, 50}.
- [V2 – Remote execution cost] The following values for *d* shall be used: 100ms, 500ms, 1000ms.

Expected results: It is expected to notice a performance increase in terms of response and completion time in case of migratory objects instead of generic objects when the remote execution cost is higher than a certain threshold.

6.2.7 Grid Producer-Consumer Objects

Purpose: The purpose of this experiment is to observe the performance aspects for producer-consumer objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe.
- [S2 - Creation] One of the *p* processes creates a generic and a producer-consumer grid object in the grid universe with ids *oid1* and *oid2*. Both objects expose a remote method *rm* that requires *d* ms to execute.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] For each object type, a producer process which continuously acquires the object exclusively and modifies its state every *d* ms. A set a *x* consumer processes perform a series of acquire operations every *d-delta* ms (experimentally selected so that a stable system is obtained). The consumer applications shall have a uniform deployment distribution in the grid universe. The produce-consume operations are run in a predefined number of cycles (e.g. 100).

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- OST – Object synchronization time [ms]
- TT – Token obtaining time [ms]
- AQT , AQET – Acquire and acquire exclusive time [ms]
- AQESR, AQSR – Acquire and acquire exclusive success rate [%]
- TRAQ, TRAEQ – Throughput for acquire and acquire exclusive [operations/s]

Experiment variants:

- [V0 – Default Node count] The number of nodes within universes should be as high as possible (e.g. depending on available hardware configuration).
- [V1 – Replication variants]: one object per universe, one object per node.

- [V2 –Frequency and cost] Selected acquire frequencies and object weights shall be applied (*d* parameter), so that a transition from a stable to unstable system is experienced.
- [V3 – Consumers] The consumer number shall be 1, *n/3*, *n/2*, *2n/3* and *n*.

Expected results: It is expected to notice a performance increase in terms of response and completion time in case of producer-consumer objects instead of generic objects if replication mechanisms are used. If no replication rules are used, there should be no performance difference as there is a unique copy of an object type in the grid universe.

6.2.8 Grid Read-Mostly Objects

Purpose: The purpose of this experiment is to observe the performance aspects for read-mostly objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of *p* = *m x n* processes are deployed, one on every grid node within the grid universe.
- [S2 - Creation] One of the *p* processes creates a generic and a read-mostly grid object in the grid universe with ids *oid1* and *oid2*. Both objects expose a remote method *rm* that requires *d* ms to execute.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] For each object type, there is a number of *pp* producer processes which continuously acquire the object exclusively and modify its state every *d* ms. A set a *cp* consumer processes perform a series of acquire operations and check if there is a change in the object state every *d-delta* ms (experimentally selected so that a stable system is obtained). The produce-consume operations are run in a predefined number of cycles (e.g. 100).

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- OST – Object synchronization time [ms]
- TT – Token obtaining time [ms]
- AQT , AQET – Acquire and acquire exclusive time [ms]
- AQESR, AQSR – Acquire and acquire exclusive success rate [%]
- TRAQ, TRAEQ – Throughput for acquire and acquire exclusive [operations/s]

Experiment variants:

- [V0 – Default Node count] The number of nodes within universes should be as high as possible (e.g. depending on available hardware configuration).
- [V1 – Replication variants]: one object per universe, one object per node.

- [V2 –Frequency] Selected acquire frequencies shall be applied (*d* parameter) so that a transition from a stable to unstable system is experienced.
- [V3 – Readers-Writes] The number of producer and consumer processes (*pp* and *cp*) should be: {*1, n/2*}, {*1, n*}, {*n/10, n/2*}, {*n/10, n*}, {*n/5, n/2*}, {*n/5, n*}, {*n/3, n/2*}, {n/3, n}.

Expected results: It is expected to notice a performance increase in terms of response and completion time in case of producer-consumer objects instead of generic objects if replication mechanisms are used. If no replication rules are used, there should be no performance difference as there is a unique copy of an object type in the grid universe.

6.2.9 Grid Result Objects

Purpose: The purpose of this experiment is to evaluate the performance aspects for result objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of *p* = *m x n* processes are deployed, one on every grid node within the grid universe.
- [S2 - Creation] One of the *p* processes creates a generic and a result grid object in the grid universe with ids *oid1* and *oid2*. Each objects type is composed out of *op* parts. Each object part is accessed via a remote method *rm* that requires *h* ms to execute.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] For each object type, there are a number of o*p* writer processes which acquire the object exclusively and modify one object part. A set a *cp* reader processes perform a series of acquire operations and read the object state. The produce-consume operations are run in a predefined number of cycles (e.g. 100).

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- OST – Object synchronization time [ms]
- TT – Token obtaining time [ms]
- AQT , AQET – Acquire and acquire exclusive time [ms]
- AQESR, AQSR – Acquire and acquire exclusive success rate [%]
- TRAQ, TRAEQ – Throughput for acquire and acquire exclusive [operations/s]

Experiment variants:

- [V0 – Default Node count] The number of nodes within universes should be as high as possible (e.g. depending on available hardware configuration).
- [V0 – Default remote execution cost] The following value for *h* shall be used: 1000ms.
- [V0 – Replication variants]: one object per node.

- [V1 – Readers-Writes] The number of reader and writer processes (consequently the number of object parts) should be: {*n/2, 1*}, {*n, 1*}, {*n/2, n/10*}, {*n, n/10*}, {*n/2, n/5*}, {*n, n/5*}, {*n/2, n/3*}, {*n, n/3*}.

Expected results: It is expected to notice a performance increase in terms of response and completion time in case of result objects instead of generic objects if replication mechanisms are used. If no replication rules are used, there should be no significant performance difference as there is a unique copy of an object type in the grid universe.

6.2.10 Grid Write-Mostly Objects

Purpose: The purpose of this experiment is to observe the performance aspects for write-mostly objects.

Workflow: The experiment consists of the following steps:

- [S1 - Deployment] A number of $p = m \times n$ processes are deployed, one on every grid node within the grid universe.
- [S2 - Creation] One of the *p* processes creates a generic and a write-mostly grid object in the grid universe with ids *oid1* and *oid2*. Both objects expose a remote method *rm* that requires *h* ms to execute.
- [S3 - Steady state] Each process out of the *p* processes issues a search operation for both *oid1* and *oid2* in order that objects are replicated and a steady state is reached.
- [S4 - Measurement] For each object type, there is a number of *pp* producer processes which continuously acquire the object exclusively and modify its state every *d* ms. A set a *cp* consumer processes perform a series of acquire operations and check if there is a change in the object state every *d-delta* ms. The produce-consume operations are run in a predefined number of cycles (e.g. 100).

Performance evaluation criteria: The following evaluation criteria should be measured:

- RT – Response time for accessing the remote object [ms]
- CT – Completion time for the application [ms]
- OST – Object synchronization time [ms]
- TT – Token obtaining time [ms]
- AQT , AQET – Acquire and acquire exclusive time [ms]
- AQESR, AQSR – Acquire and acquire exclusive success rate [%]
- TRAQ, TRAEQ – Throughput for acquire and acquire exclusive [operations/s]

Experiment variants:

- [V0 – Default Node count] The number of nodes within universes should be as high as possible (e.g. depending on available hardware configuration).
- [V0 – Default remote execution cost] The following value for *h* shall be used: 1000ms.
- [V1 – Replication variants]: one object per universe, one object per node.
- [V2 –Frequency] Selected acquire frequencies shall be applied (*d* parameter).

- [V3 – Readers-Writes] The number of producer and consumer processes (*pp* and *cp*) should be: {*n/2, 1*}, {*n, 1*}, {*n/2, n/10*}, {*n, n/10*}, {*n/2, n/5*}, {*n, n/5*}, {*n/2, n/3*}, {*n, n/3*}.

Expected results: It is expected to notice a performance degradation once a certain producer-consumer ratio is exceeded.

6.3 Theoretical System Analysis

This section describes a brief theoretical analysis of the grid shared data model for the experiments described in Section 6.2. The analysis is based on the cost model introduced in previous chapters which defines the communication costs in the grid universe. The cost model considers a homogeneous system, where a number of *m* universes are considered, and each universe containing a number of *n* nodes. The communication medium is error free, thus all communications between different entities have a certain upper limit.

Let φ denote the communication cost between any two universes. φ is typically expressed in time units which denote the round-trip time to send a message and receive the response of the message. In other words, any node that sends a message to another node residing in a different universe has a cost equal to φ. Let δ denote the communication cost between any two nodes within the same universe. δ is typically expressed in time units which denote the round-trip time to send a message and receive the response of the message. In other words, any node that sends a message to another node residing in the same universe has a cost equal to δ.

The theoretical analysis applies to a stable system which in this case denotes the absence of any queuing effects, as if there is only one request at any given time. As seen in the grid universe model, several message queues are present in the system. In this analysis, the queues are not considered meaning that a stable system is assumed where message queues are not inducing any communication latencies and the message arrival frequency equals the message departure frequency. The following notations are used:

- *GU* – grid universe
- *U* - universe
- *PN* – primary node
- $RT_{OP}(C,N)_n^m$ - response time for operation *OP* that is issued from client *C* on node *N* in the grid universe with *m* universes each containing *n* nodes.

If we refer to the cost as time units, and consider a communication between a client *C* and a node *N*, we note:

$$RT_{OP}(C,N)_n^m \le \varphi, where: C \in U1 \wedge N \in U2, U1 \in GU, U2 \in GU \wedge U1 \cap U2 = \Phi$$

The communication between nodes that belong to the same universe has an upper limit δ. If we refer to the cost as time units, and considering a communication between a client *C* and a node *N*, we note:

$$RT_{OP}(C,N)_n^m \le \delta, where: C \in U \wedge N \in U$$

In the following sections, a particular configuration of the model is considered that reflects a hypothetical deployment. If not specified explicitly, the following parameters are used in the analysis:

$$\delta = 20ms, \varphi = 100ms, n = 10, m = 5$$

6.3.1 Grid Object Search

In the experiment described in Section 6.2.1, the search response time is evaluated when different replication rules are used. The search algorithm implies that when an object is looked up, the primary node within the universe where the search is issued from is contacted. If the object is registered there, the node where the object resides is returned. In the generic case where no replication is used, if the object is not present in that universe, the object is looked-up on every known universe until it is found. Considering that for m universes and that in the worst case there are up to m-1 search redirections, the following response time is obtained:

$$RT_{Search}(C,N)_n^m \le RT(C,PN) + \sum_{\substack{1 \le P \prec m \\ P \ne PN}} RT(P,PN) \le \delta + (m-1)\varphi \qquad (1)$$

In case “one object per universe” replication rule is used, there will be always a replica of any object in the local universe and

$$RT_{Search}(C,N)_n^m \le RT(C,PN) \le \delta \qquad (2)$$

In case “one object per node is used” replication rule is used, although there will be always a replica of any object on the client node, the search time implies contacting the primary node and:

$$RT_{Search}(C,N)_n^m \le RT(C,PN) \le \delta \qquad (3)$$

For example, considering the above three situations, there are two possible search response time characteristics that are depicted in Figure 43. The dotted red line represents the right hand side of formula (1) and the green line formulas (2) and (3).

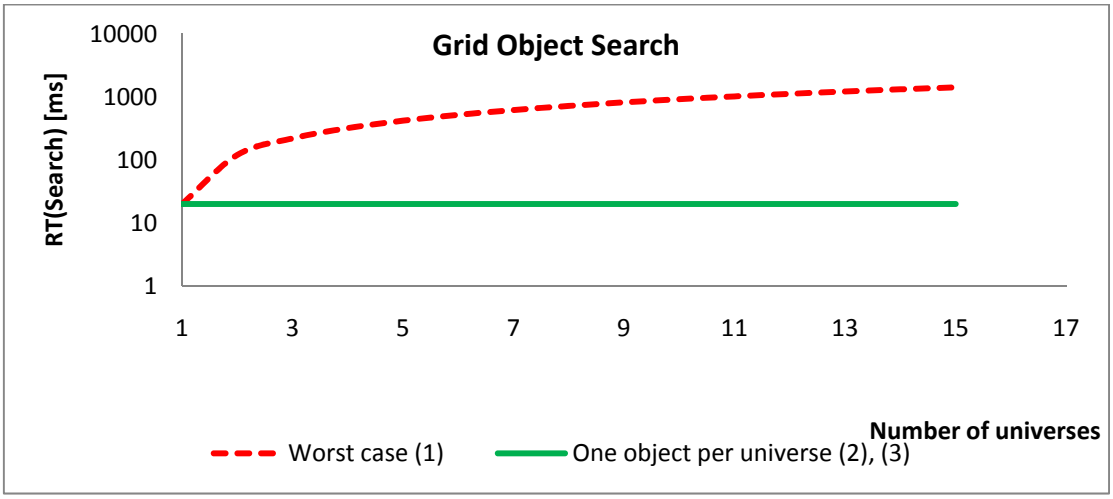

Figure 43: Grid Search Object Time

6.3.2 Acquire and Acquire Exclusive

The experiment described in Section 6.2.2 aims to highlight performance aspects of the acquire operation. The operation does not depend on the replication scheme, but rather on the token's location.

In case of the generic grid object, the node that requires access to the data needs to request access from the primary node that holds the token for that specific object type. Access is granted if there are no other acquire-exclusive requests. Assuming that the requests have to travel across the longest chain of primary nodes so that:

$$AQT_{Generic}(C,N)_n^m \le RT(C,PN) + RT_{Token}(P,PN) + TT_{Generic} \le \delta + \log(m)\varphi + TT_{Generic} \quad (4)$$

where $TT_{Generic}$ represents the local token obtaining time. Basically $TT_{Generic}$ refers to the time required to obtain access to a resource assuming that the token is held by the node that requires the resource. $TT_{Generic}$ depends on the local resource arbitration algorithm. It also depends on the number of waiting requests to be processed when a request is issued, namely the pending acquire exclusive and non-exclusive requests. Typically it includes also the resource holding time by all pending requests. As this value is application specific, it can be only measured or simulated.

In case of the generic object, the acquire time does not depend on the object distribution, but rather on the token distribution. If the token is held by the universe where *C* resides, the acquire time becomes:

$$AQT_{Generic}(C,N)_n^m \le RT(C,PN) + TT \le \delta + TT_{Generic} \quad (5)$$

Formula (5) represents the ideal case where all requests are made within the same universe and represents a lower bound for AQT. In case the token resides in another universe, according to the Grid Universe model, there can be up to m-1 possible token requests within the grid universe, which leads to an upper bound for AQT. Considering both situations, the following formula is obtained:

$$\delta + TT_{Generic} \le AQT_{Generic}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{Generic} \quad (6)$$

Assuming that no acquire-exclusive requests have been made, it means that $TT_{Generic}$ can be neglected since the token neither has to migrate, nor to be held by a node. Figure 44 reflects formula (7) where $TT_{Generic} = 20ms$

$$\delta \le AQT_{Generic}(C,N)_n^m \le \delta + \log(m)\varphi \quad (7)$$

In case of the acquire exclusive operation, in addition to the operations reflected in (7), the token migration and potential object synchronization time must be reflected so that:

$$AQET_{Generic}(C,N)_n^m \le RT(C,PN) + RT_{Token}(P,PN) + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \le \\ \le \delta + \log(m)\varphi + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \quad (8)$$

As discussed in this section, the three parameters are application dependent and they can be simulated or measured. The token migration time as well as the synchronization time is directly dependent on φ and the size of the token representation respectively the size of the grid shared object.

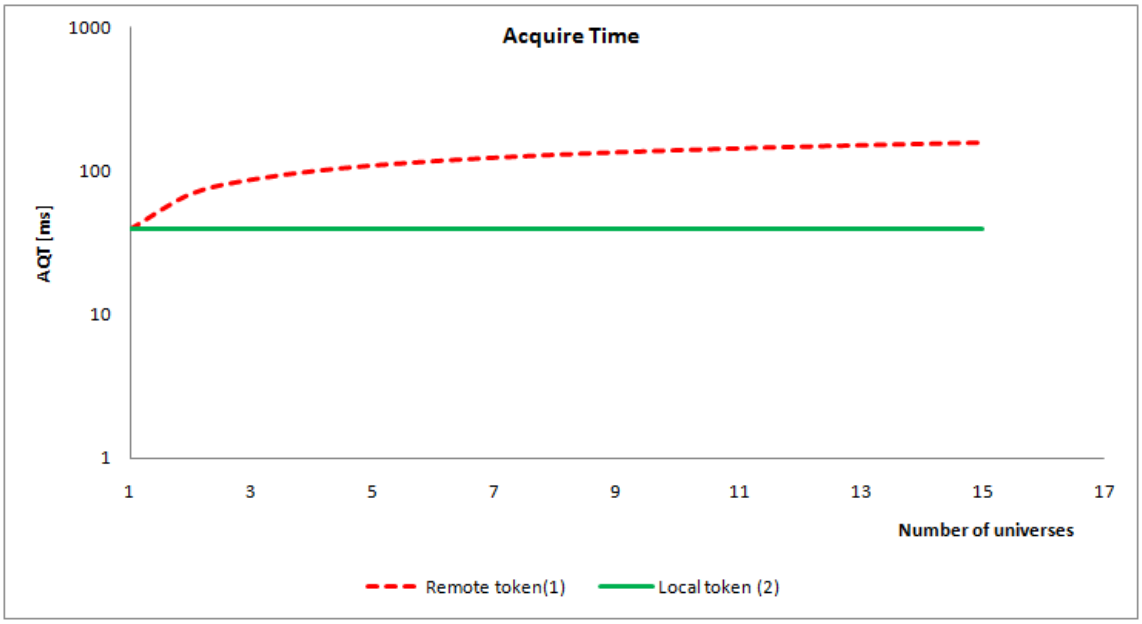

Figure 44: Acquire Time

6.3.3 Grid Read-Only Objects

The experiment described in Section 6.2.4, aims to highlight performance aspects of grid read-only objects compared to the situation where generic grid objects are used. First, in the experiment definition, the response time for search operation is considered. As seen in section 6.3.1, the search time depends only on the replication scheme which results in the object distribution in the grid universe. Considering that read-only objects are using an aggressive replication rule ("one object per node" if the size of the object is reasonably small), there are no differences in terms of the search response time compared to generic grid objects provided that the same replication rules are used for generic objects. In the analysis of the *acquire time AQT* we have the following situations, depending on the object distribution:

(C1). "One object per node" replication

$$AQT_{Read-Only}(C,N)_n^m = 0 \qquad (9)$$

as there is always a replica on *N*.

(C2). "One object per universe" replication and C and N belong to the same universe

$$AQT_{Read-Only}(C,N)_n^m \le \delta \qquad (10)$$

as a call to N is necessary to arbitrate the data access.

(C3). "One object per universe" replication and C and N belong to the different universe

$$AQT_{Read-Only}(C,N)_n^m \le \varphi \qquad (11)$$

as a call to N is necessary to arbitrate the data access. This situation is impossible to occur since read-only object are there will be exactly one replica of any read-only object in any universe.

The response time for reading object data depends at the end on the replication scheme which is used. Similar to the search operation analysis, if the same replication rules are used, object reading time shall be the same for read-only and generic grid objects. In both situation (assuming that acquire exclusive requests are not issued so that timeouts cannot occur), the acquire success rate for both object types is expected to be 100%.

Obviously, the acquire time for read-only objects is by far lower than the one for the generic objects, thus a significant performance increase would be expected in case of the usage of read-only objects. Considering the application completion time, in case of the generic objects we have the following situation:

$$\begin{aligned} CT_{Generic}(C,N)_n^m &= AQT_{Generic}(C,N)_n^m + \mathrm{Re}\,ad_{Generic} + \mathrm{Re}\,lease_{Generic}(C,N)_n^m + d \\ CT_{Generic}(C,N)_n^m &\le \delta + \log(m)\varphi + \mathrm{Re}\,ad_{Generic} + \varphi + \delta + d \\ CT_{Generic}(C,N)_n^m &\le 2\delta + (1+\log(m))\varphi + \mathrm{Re}\,ad_{Generic} + d \end{aligned} \quad (12)$$

In case of read-only objects:

$$\begin{aligned} CT_{\mathrm{Re}\,ad-Only}(C,N)_n^m &= AQT_{\mathrm{Re}\,ad-Only}(C,N)_n^m + \mathrm{Re}\,ad_{\mathrm{Re}\,ad-Only} + \mathrm{Re}\,lease_{\mathrm{Re}\,ad-Only}(C,N)_n^m + d \\ CT_{\mathrm{Re}\,ad-Only}(C,N)_n^m &\le \delta + \mathrm{Re}\,ad_{\mathrm{Re}\,ad-Only} + \delta + d \\ CT_{\mathrm{Re}\,ad-Only}(C,N)_n^m &\le 2\delta + \mathrm{Re}\,ad_{\mathrm{Re}\,ad-Only} + d \end{aligned} \quad (13)$$

A graphical representation of formulas (12) and (13) is shown in Figure 45, considering the same value for the read operation, as if the object was located on the same node. As expected, a better performance is obtained by using grid read-only objects instead of the generic objects. In addition, the application performance is expected to be independent on the number of deployed nodes and universes.

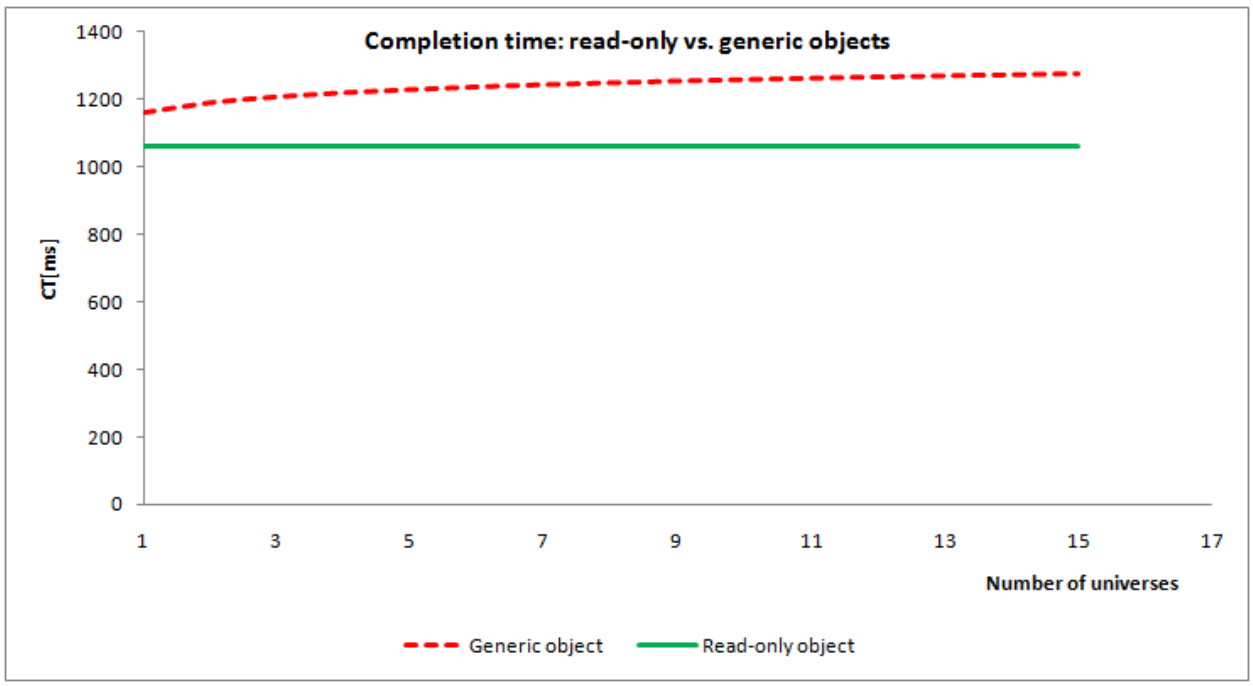

Figure 45: Grid Read-Only Objects – Application Completion Time

6.3.4 Grid Private Objects

The experiment described in Section 6.2.5, aims to highlight performance aspects of grid private objects compared to generic grid objects. Similar to the case of read-only objects as seen in the previous section, the search time depends on the object replica distribution. As private objects are not replicated, generic grid objects might have an advantage due to replication in terms of the search operation. The worst case scenario is identical for the two situations when "one object per universe" is used for generic grid objects. If "one object per node" replication rule is used, search time for the generic grid object becomes $RT_{Search}(C,N)_n^m \le RT(C,PN) \le \delta$ opposite to $RT_{Search}(C,N)_n^m \le \delta + (m-1)\varphi$ which is required for private object search.

The ***acquire time AQT or AQET*** is determined directly by the communication cost to the node where the data resides as the token mechanism is avoided. This means that basically AQT and AQET depends only if the node holding the private objects resides or not in the same universe as the client node as following (considering in a stable system):

(C1). C and N belong to the same universe

$$AQT_{\mathrm{Private}}(C,N)_n^m \le \varphi + TT_{\mathrm{Private}} \qquad (14)$$

(C2). C and N belong to different universes

$$AQT_{\mathrm{Private}}(C,N)_n^m \le \delta + TT_{\mathrm{Private}} \qquad (15)$$

where *TT* represents the token obtaining time. *TT* refers to the time required to obtain access to a resource assuming that the token is held by the node that requires the resource. It depends on the local resource arbitration algorithm and of course the number of the requests waiting to be processed when a request is issued. It also contains the resource holding time by all pending requests. As this value is implementation and application specific, it can be only measured or simulated.

In case of the generic grid object, we have the same situation as described in (6) and (8).

Due to algorithmic complexities, it is expected that $TT_{\mathrm{Private}} \le TT_{Generic}$ meaning that $AQT_{\mathrm{Private}}(C,N)_n^m < AQT_{Generic}(C,N)_n^m$ no matter what replication rule is used for the generic objects.

The response time for reading object data depends on the size of the object as well as on the object's location relative to the caller node. Similar to the search operation analysis, if the same replication rules are used (e.g. no replication), object reading time shall be the same for private and generic grid objects. In this case, the application completion time becomes:

$$\begin{aligned} CT_{\mathrm{Private}}(C,N)_n^m &= AQT_{\mathrm{Private}}(C,N)_n^m + \mathrm{Read}_{\mathrm{Private}} + \mathrm{Release}_{\mathrm{Private}}(C,N)_n^m + d \\ CT_{\mathrm{Private}}(C,N)_n^m &\le \delta + TT_{\mathrm{Private}} + \delta + d \\ CT_{\mathrm{Private}}(C,N)_n^m &\le 2\delta + TT_{\mathrm{Private}} + d \end{aligned} \qquad (16)$$

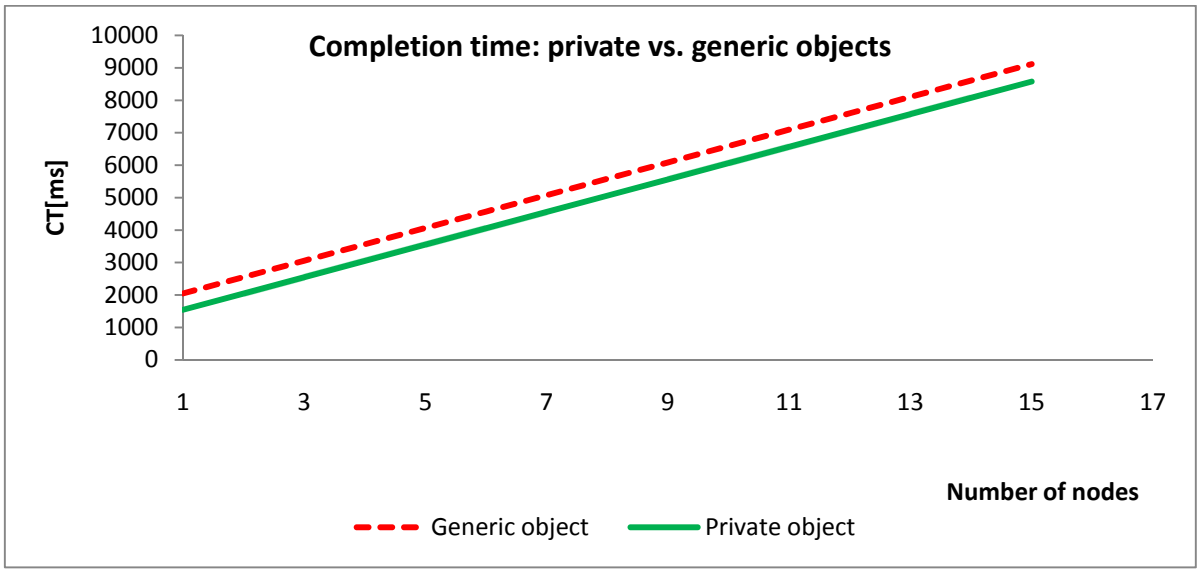

Figure 46: Grid Private Objects – Application Completion Time

If we consider for example that an average number of n/2 clients are waiting for the token at a given time and token request handling takes 5ms for the private objects and 10ms for generic objects (due to algorithm complexity), formulas (18) and (14) become the following, which is drawn in Figure 46:

$$TT_{Private} = \frac{n}{2}(10+d)$$

$$TT_{Generic} = \frac{n}{2}(5+d)$$

$$CT_{Private}(C,N)_n^m \le 2\delta + \frac{n}{2}(5+d) + d \qquad (17)$$

$$CT_{Generic}(C,N)_n^m \le 2\delta + m\varphi + \frac{n}{2}(10+d) + d \qquad (18)$$

6.3.5 Grid Migratory Objects

The experiment described in Section 6.2.6 aims to highlight performance aspects of grid migratory objects compared to generic grid objects. Similar to the case of read-only objects as seen in section 6.3.1, the search time depends on the object replica distribution. As migratory objects are not replicated, generic grid objects might have an advantage due to replication in terms of the search operation. The worst case scenario is identical for the two situations when “one object per universe” is used for generic grid objects. If “one object per node” replication rule is used, search time for the generic grid object becomes $RT_{Search}(C,N)_n^m \le RT(C,PN) \le \delta$ opposite to $RT_{Search}(C,N)_n^m \le \delta + (m-1)\varphi$ which is required for migratory object search.

The ***acquire time AQT or AQET*** is determined by the search time due to resolving invalid references. As a consequence, there are four possible situations:

(C1). If the reference is valid and C and N belong to same universes

$$AQET_{Migratory}(C,N)_n^m \le \delta + TT_{Migratory} \quad (19)$$

(C2). If the reference is valid and C and N belong to different universes

$$AQET_{Migratory}(C,N)_n^m \le \varphi + TT_{Migratory} \quad (20)$$

(C3). If the reference is invalid and C and N belong to different universes

$$AQET_{Migratory}(C,N)_n^m \le RT_{Search}(C,N)_n^m + \varphi + TT_{Migratory} \le \delta + m\varphi + TT_{Migratory} \quad (21)$$

(C4). If the reference is invalid and C and N belong to the same universe

$$AQET_{Migratory}(C,N)_n^m \le RT_{Search}(C,N)_n^m + \delta + TT_{Migratory} \le 2\delta + (m-1)\varphi + TT_{Migratory} \quad (22)$$

where *TT* represents the token obtaining time. *TT* refers to the time required to obtain access to a resource assuming that the token is held by the node that requires the resource. It depends on the local resource arbitration algorithm and of course the number of the requests waiting to be processed when a request is issued. It also includes the resource holding time by all pending requests. As this value is application specific, it can be only measured or simulated.

In case of the generic grid object, we have the same situation as shown in section 6.3.1 in formulas (6) and (8), where both an upper and lower bounds can be defined depending on the token location with respect to the node holding the shared data:

$$\delta + TT_{Generic} \le AQT_{Generic}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{Generic} \quad (6)$$

$$AQET_{Generic}(C,N)_n^m \le RT(C,PN) + RT_{Token}(P,PN) + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \le \le \delta + \log(m)\varphi + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \quad (8)$$

Based on (22) and (8), $AQET_{Migratory}(C,N)_n^m \le AQET_{Generic}(C,N)_n^m \Leftrightarrow$

$$2\delta + (m-1)\varphi + TT_{Migratory} \le \delta + \log(m)\varphi + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \Leftrightarrow$$

$$\delta + (m-1)\varphi + TT_{Migratory} \le \log(m)\varphi + TT_{Generic} + TT_{Migrate} + SYNC_{Obj} \quad (23)$$

In case of the experiment considered in 0 every access to the migratory objects will result in an invalid reference which needs to be resolved. Considering every two consecutive accesses from two different nodes, there are two situations depending on the relative node's locations: C3 and C4. It is important to note that at the global scope, where multiple acquire exclusive requests are being issued, the acquire time for migratory objects depends heavily on the performance of the search operation. Under these considerations, the situations C3 and C4 become:

(C3'). If the reference is invalid and C and N belong to different universes

$$AQET_{Migratory}(C,N)_n^m \le \delta + m\varphi + TT_{Migratory} \Leftrightarrow$$

$$AQET_{Migratory}(C,N)_n^m \le \delta + m\varphi + SYNC_{Object} + \delta = 2\delta + m\varphi + SYNC_{Object} \quad (24)$$

$$AQET_{Generic}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{Generic} + TT_{Migrate} + SYNC_{Object} \Leftrightarrow$$

$$AQET_{Generic}(C,N)_n^m \le 2\delta + \log(m)\varphi + TT_{Generic} + SYNC_{Object} \quad (25)$$

Considering a stable system where $TT_{Generic}$ is negligible, based on (24) and (25) it can be observed that between universes: $AQET_{Generic}(C,N)_n^m \le AQET_{Migratory}(C,N)_n^m \Leftrightarrow \log(m)\varphi \le m\varphi$

(C4'). If the reference is invalid and C and N belong to the same universe

$$AQET_{Migratory}(C,N)_n^m \le \delta + TT_{Migratory} \Leftrightarrow$$

$$AQET_{Migratory}(C,N)_n^m \le \delta + SYNC_{Object} + \delta = 2\delta + SYNC_{Object} \quad (26)$$

$$AQET_{Generic}(C,N)_n^m \le \delta + TT_{Generic} + SYNC_{Object} \quad (27)$$

Considering a stable system where $TT_{Generic}$ is negligible, based on (26) and (27) it can be noticed that within a universe $AQET_{Generic}(C,N)_n^m \le AQET_{Migratory}(C,N)_n^m \Leftrightarrow \delta \le 2\delta.$

The response time for reading object data depends at the end on the replication scheme that is used. Similar to the search operation analysis, if the same replication rules are used (e.g. no replication), object reading time shall be the same for migratory and generic grid objects. A major advantage for migratory objects is when a high number of operations are issued that requires a high amount of data marshaling across the network. In these cases, the locality characteristics of the migratory objects can play a significant role and decrease the total application execution time.

6.3.6 Grid Producer-Consumer Objects

The experiment described in Section 6.2.7 aims to highlight performance aspects of grid producer-consumer objects compared to generic grid objects. Similar to the case of previous types of objects and as seen in section 6.3.1, the search time depends on the object replica distribution. As producer-consumer objects are replicated in the same way as generic grid objects, there is no difference in search time and formulas (1), (2) and (3) apply in this situation too.

In terms of the ***acquire time AQT or AQET*** , there is no difference compared to the generic grid object and both lower and upper bounds apply in the same manner as follows:

$$\delta + TT_{PC} \le AQT_{PC}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{PC} \quad (28)$$

Generally speaking, the token time is composed out of three different operations execution times:

$$TT_{Generic} = Token\,\mathrm{Re}quest + ClientOperation + ObjectSynchronization \quad (29)$$

In case of producer-consumer objects, where an eager synchronization protocol is used, the object synchronization time overlaps with the client operation, thus:

$$TT_{PC} = Token\,\mathrm{Re}quest + ClientOperation \quad (30)$$

As seen from (29) and (30), the difference in AQT between the generic grid objects and producer-consumer objects is reduced in the object synchronization time. The object synchronization time depends on the number of replicas that need to be synchronized and the size of the object. Depending on these

situations a performance increase up to the object synchronization time can be expected for acquire operations. It means that under any circumstances $AQT_{PC}(C,N)_n^m \leq AQT_{Generic}(C,N)_n^m$

In terms of the release time, in case of the generic objects the release time is reduced to the penalty of sending a message to the primary node from the node holding the object lock. However, in case of the producer consumer objects, besides notifying the primary node, an object synchronization protocol is followed where all consumer objects that are currently in use are being updated eagerly. This means that:

$$RLT_{PC}(C,N)_n^m > RLT_{Generic}(C,N)_n^m \quad (31)$$

Considering the acquire exclusive time, in case of the producer-consumer object where there is only one producer node, the acquire exclusive time is expected to be constant (assuming that a stable system is observed). For producer-consumer objects there is no object synchronization necessary in the acquire exclusive operation since there is only one producer which acquires the object from time to time, thus $AQET_{PC}(C,N)_n^m \leq AEQT_{Generic}(C,N)_n^m$ (32)

6.3.7 Grid Read-Mostly Objects

The experiment described in Section 6.2.8 aims to highlight performance aspects of grid read-mostly objects compared to generic grid objects. Similar to the case of producer-consumer objects and as seen in section 6.3.1, the search time depends on the object replica distribution. As read-mostly objects are replicated in the same way as generic grid objects, there is no difference in search time and formulas (1), (2) and (3) apply in this situation too.

In terms of the ***acquire time AQT or AQET*** , there is no difference compared to the generic grid object and both lower and upper bounds apply in the same manner as follows:

$$\delta + TT_{RM} \leq AQT_{RM}(C,N)_n^m \leq \delta + \log(m)\varphi + TT_{RM} \quad (33)$$

Similar to producer-consumer objects, an eager synchronization protocol is used and the object synchronization time might partially overlap with the client operation, thus:

$$TT_{RM} = TokenRq + ClientOperation \quad (34)$$

In general it is expected that up to a certain number of writers and write frequencies acquire time for read-mostly objects would be less than for generic objects. Since the performance gain is determined by the total overlapped operations (synchronization and client operations), it is impossible to express exactly the difference to generic objects.

The only difference between read-mostly objects and producer-consumer objects is in the number of writers (or producers). The difference in terms of writer count can be reflected in the object synchronization protocol where the identity of readers and writers is more difficult to manage in case that multiple writers are allowed. It means that potentially the token time for read-only objects would be higher that the token time for producer-consumers and the probability of complete overlapping between object state synchronization and object usage is smaller. A quantitative definition of this overlapping would require a probabilistic estimation approach in the theoretical analysis which is aimed to be covered by prototype experiments as well as through computer aided verification support.

6.3.8 Grid Result Objects

The experiment described in Section 6.2.9 aims to highlight performance aspects of grid result objects compared to generic grid objects. Similar to the case of previous types of objects and as seen in section 6.3.1, the search time depends on the object replica distribution. As result objects are replicated in the same way as generic grid objects, there is no difference in search time and formulas (1), (2) and (3) apply in this situation too.

In terms of the ***acquire time AQT*** , there is no difference compared to the generic grid object and both lower and upper bounds apply in the same manner:

$$\delta + TT_{\mathrm{Re}\,sult} \le AQT_{\mathrm{Re}\,sult}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{\mathrm{Re}\,sult} \quad (35) \text{ and}$$

$$TT_{\mathrm{Re}\,sult} = Token\ \mathrm{Re}\,quest + ClientOper\ ation + ObjectSync\ hronizatio\ n \quad (36)$$

In case of result objects, object synchronization protocol is more complex than in case of generic grid objects. First, the latest objects parts are identified and then the state of the target object is assembled out of the distributed object states. Depending on object part distribution and their sizes different synchronization time values can be experienced. An analytical analysis of these aspects would require an extension towards probabilistic models which is exceeds the scope of the static analysis. The overall performance differences are aimed to be covered in later sections on prototype experiments as well as computer aided analysis.

In terms of the ***acquire exclusive time AQET*** , each acquire exclusive operation does not lock the object part, but only checks if the grid object can be accessed (e.g. no other client holds the object in exclusive mode). As a result:

$$\delta \le AQET_{\mathrm{Re}\,sult}(C,N)_n^m \le \delta + \log(m)\varphi \quad (37)$$

Thus, it is expected that AQET for result objects to be much smaller than in case of generic objects due to the missing synchronization lock.

6.3.9 Grid Write-Mostly Objects

The experiment described in Section 6.2.10 aims to highlight performance aspects of grid write-mostly objects compared to generic grid objects. Similar to the case of previous types of objects and as seen in section 6.3.1, the search time depends on the object replica distribution. As write-mostly objects are replicated in the same way as generic grid objects, there is no difference in search time and formulas (1), (2) and (3) apply in this situation too.

As write-mostly objects are not different than generic objects (they carry semantic information especially for the selection of a proper replication rule), the ***acquire time AQT or AQET*** is identical to the generic grid object and both lower and upper bounds apply in the same way:

$$\delta + TT_{Write-Mostly} \le AQT_{Write-Mostly}(C,N)_n^m \le \delta + \log(m)\varphi + TT_{Write-Mostly} \quad (38)$$

The performance aspects are only affected by the replication rules and the number of readers and writes. Same as previous objects type, an analytical analysis of these aspects would require an extension towards probabilistic models which is quite complex. The overall performance differences are aimed to be covered in later sections on prototype experiments as well as computer aided analysis.

7 Prototype Analysis

This chapter describes the analysis of a prototype model of the system that was developed in order to assess various performance and qualitative aspects as described in section 6.2. First, the prototype architecture is described and then the experimental results are presented.

7.1 GUN Architecture

GUN is the acronym for **G**rid **UN**iverse and represents a Java based implementation of the grid universe model defined in [132]. Remote interactions are expressed in GUN based on Java's remote object model. First, the Remote Method Invocation (RMI) solution was chosen for its simplicity and ease of use. Second, because the system model does not require multicasting support (like Jini [129] or ProActive [130] solutions do), the RMI model fits well to the abstract model.

GUN reflects the architecture of the abstract model and the abstract system architecture described in [131]. Similar to the abstract model, in GUN there are a set of processes deployed over several networks called universe nodes. The universe nodes are homogeneous and each of them is able to accommodate a certain number of data items, until the available capacity of the universe node is consumed. Typically universe nodes are grouped together in network latency proximity and form a universe. The collection of all deployed universes forms the grid universe. Each universe contains a dedicated node called "primary node" which manages the communication with other universes and indexes the information on available data items accommodated by each node within the same universe. All primary nodes can be seen as a distributed registry, each being responsible for managing certain number of data objects.

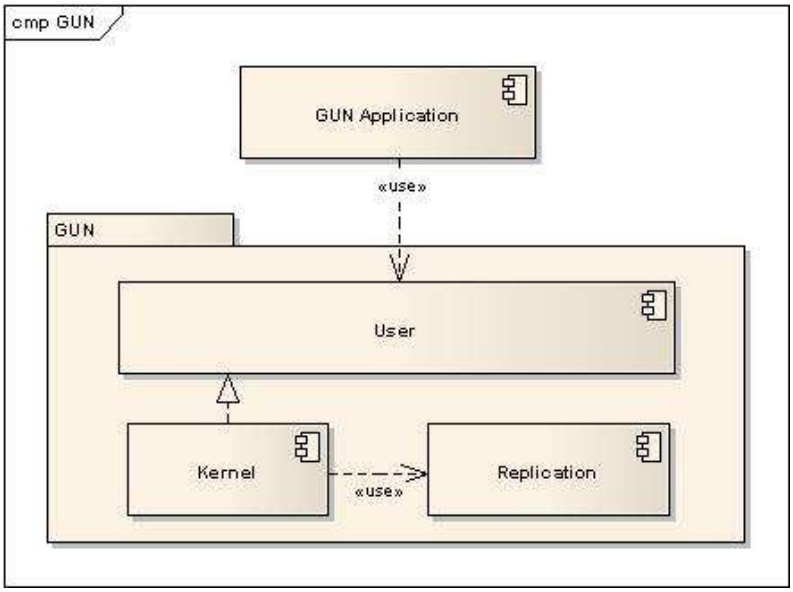

Figure 47: GUN Layers

The GUN prototype is divided into three layers, as illustrated in Figure 47. There is a user layer which exposes the abstractions and necessary interfaces to the application programmer. The second layer is the kernel which implements the core algorithms and implements all interfaces exposed to the

outside world by the user layer. Last but not least, there is a replication layer which handles object replication policies. The replication layer implements an interface required by the kernel so that the kernel invokes the replication engine at some key points in order to trigger object replication. The replication layer is extendable, meaning that user defined replication rules can be registered into the GUN architecture.

7.1.1 GUN User Layer

The user layer depicted in Figure 48 provides services to create, find, delete and acquire grid objects. The services are exposed through the *GridUniverse* class which is implemented as a singleton object. When an object is created, GUN returns a handle to that object. The handle contains information about the object identifiers OID, GID and an URI of the remote object in the RMI domain. The handle shall be passed by the client whenever an operation on the grid objects is invoked such as removal, acquire or release. Basically, the application programmer extends the *GridObject* class in order to implement its custom objects. The *GridObject* implements the RMI specific *Remote* interface, meaning that GUN user defined objects are automatically remote objects. The concrete custom interface is retrieved from GUN using the *GetGridObjectRef* method of the *GridObjectHandle* class.

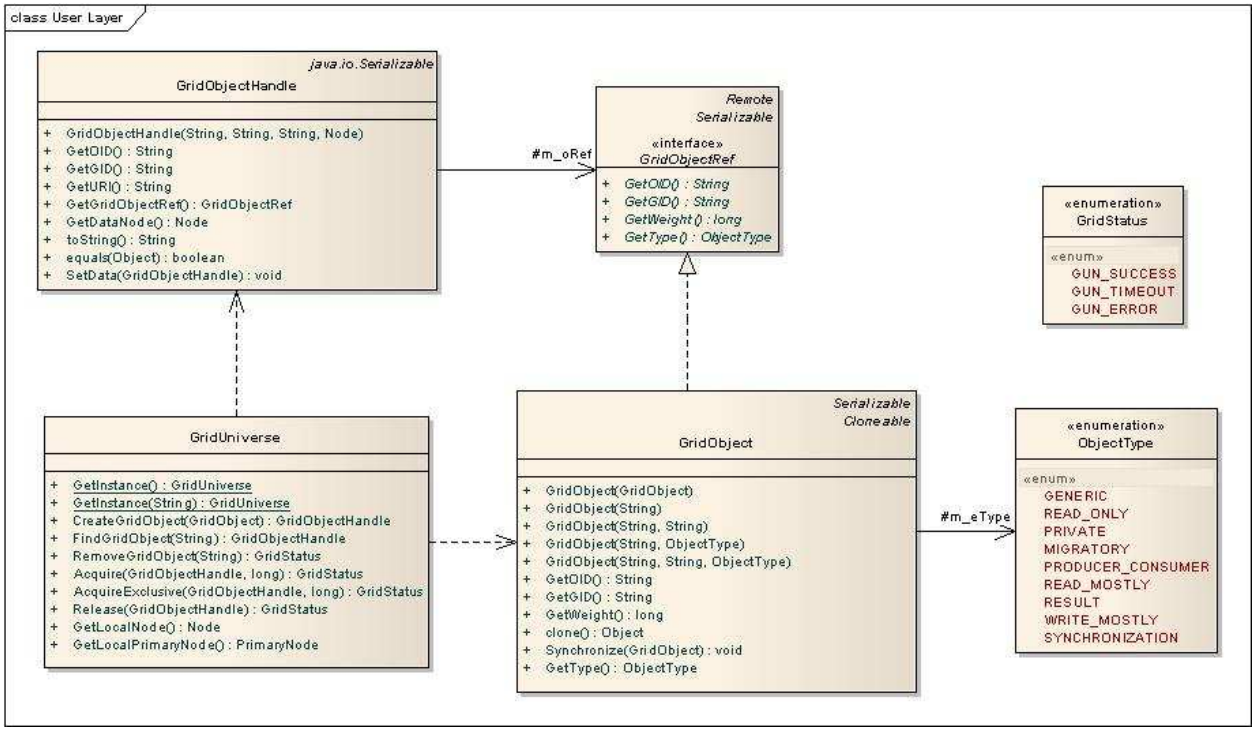

Figure 48: GUN User Layer

Object creation follows the locality principle and tries to find a node in the proximity of the node from where the request was issued (e.g. caller node). Upon completion, the create service returns a handle to the client that can be used to acquire exclusive or non-exclusive access to the object instance as well as to invoke specialized methods that are provided in the concrete object definition. Object finding follows the same data locality principle and tries to locate an object that resides in the proximity of the caller application.

7.1.2 GUN Kernel

In the ***GUN kernel***, there are interfaces defined for nodes and primary nodes as well as their implementations for grid nodes and grid primary nodes. The grid node related classes that are described in Figure 49 are not exposed by the user layer and thus they are not visible to the GUN application programmer.

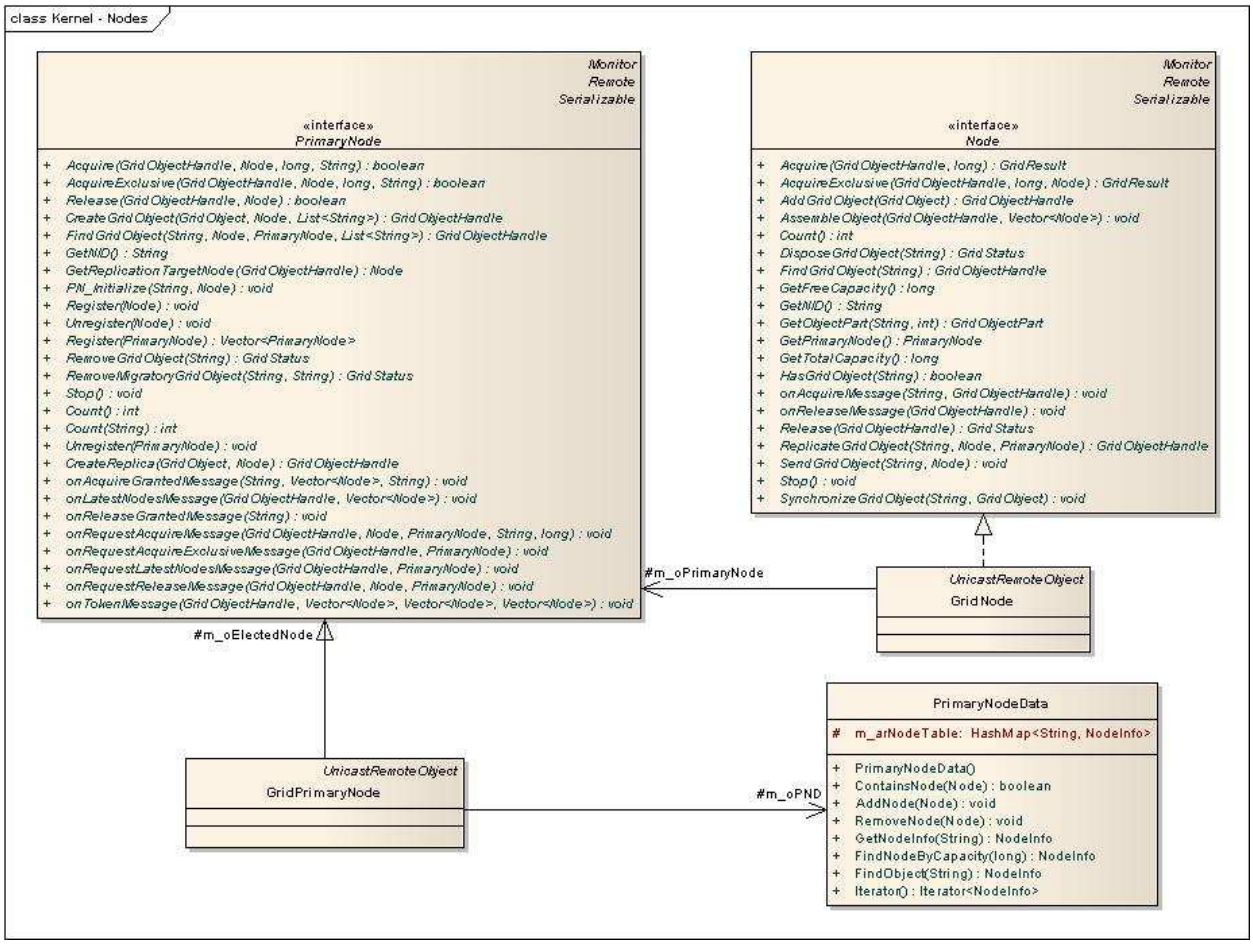

Figure 49: GUN Architecture: GridNode and GridPrimaryNode

When the GUN system is started, first all primary nodes are being started. Each primary node has a configuration file which contains the address of at least another primary node so that following the known primary nodes, all primary nodes can be eventually known to one another.. Based on this minimalistic configuration file, primary nodes are able to discover all the other primary nodes dynamically. The primary nodes are running a simple discovery protocol, which at the end brings all primary nodes to know the identity of all other primary nodes. The same mechanism is applied when a primary node is removed from the grid universe. As a result, in the GUN system, it is ensured that every primary node knows all other primary nodes, or in other terms, all universes know all other universes. This decision has been made based on the assumption that primary nodes are running on dedicated machines, which have a high availability rate (e.g. hardware fault tolerance). The GUN system can be extended from this point of view to a peer-to-peer like discovery protocol between primary nodes.

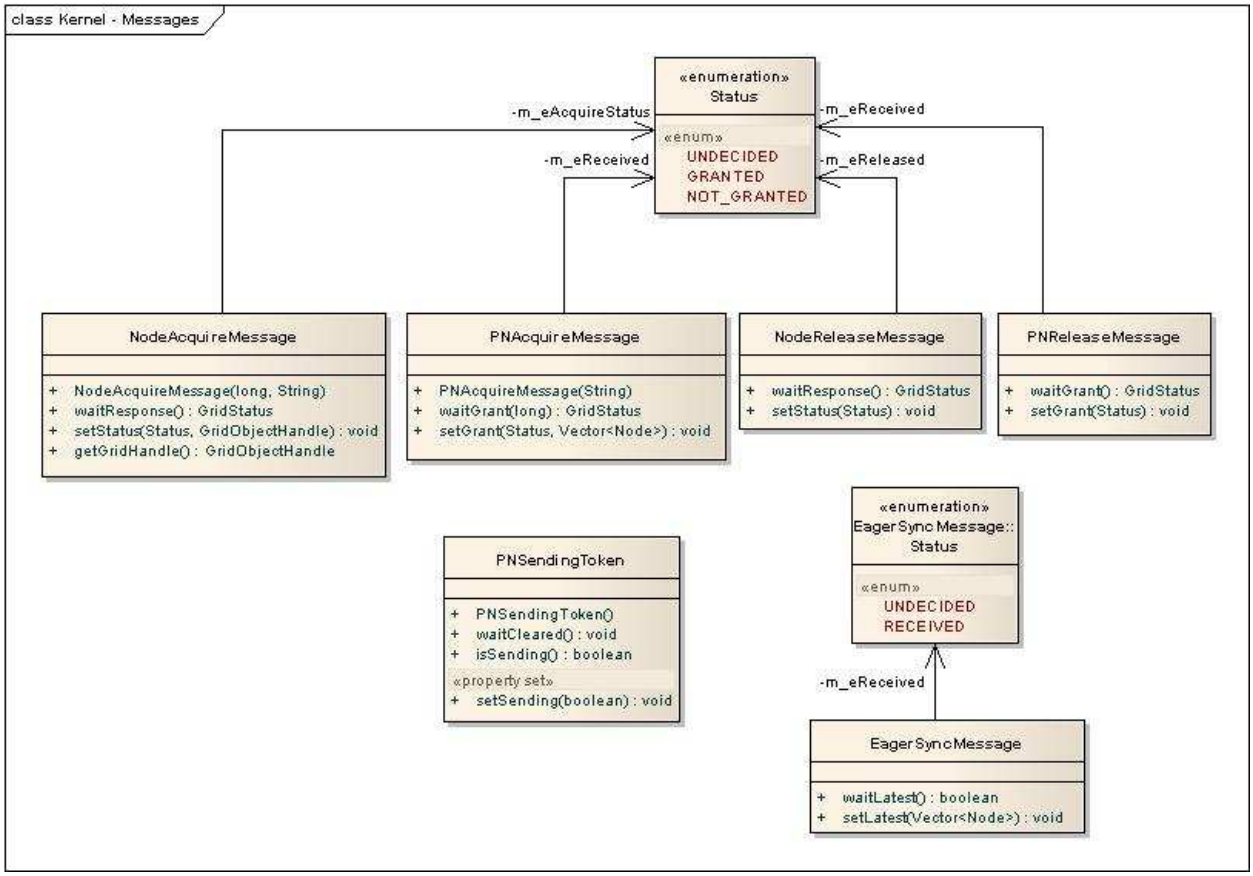

Figure 50: GUN Architecture: Kernel Messages

After the primary nodes are started, the grid nodes are deployed. Every grid node has a configuration file that specifies its name, capacity and the address of the primary node where it must register. Normally the grid nodes are located in network latency proximity, meaning that in every universe there are homogeneous communication characteristics. When the node is instantiated, it automatically registers to the designated primary node. The primary node stores information in a hash table about all the registered nodes and their status (e.g. available capacity, stored objects etc). Using a hash table mechanism it is ensured that a fast lookup time is achieved.

The kernel component implements the mutual exclusion algorithms and the model defined in [131] where an extended version of the distributed multi-token Naimi-Trehel algorithm has been defined. The interaction between nodes and primary nodes is happening via remote message invocations (RMI). This interaction follows the following pattern: request messages are sent via methods named like *DoSomething()* while callbacks are received via methods named *onSomeMessage().* Internally, the asynchronous communication is realized via message classes that are described in Figure 50.

7.1.3 GUN Mutual Exclusion Handling

When a client invokes an operation on a grid object via the GridUniverse, the node where the client resides gets invoked one of the following methods (depending on the desired operation): Acquire, AcquireExclusive or Release. The node creates the corresponding request message and adds it into its request queue. Next, the node delegates the operation to the primary node to which it had registered.

Next, the node is waiting for the primary node to reply to its request by calling a wait method on the queued message. After processing the node's request, the primary node responds to the node by calling one of the callback methods which triggers a notification on the awaited message. After the node is notified by the awaited message, the message is removed from the queue, the original client method invocation ends and the response is returned to the client. This mechanism is used for all interactions between nodes and primary nodes.

The interaction between grid primary nodes is more complex and it basically implements the multi-token Naimi-Trehel algorithm. All requests that are sent by grid nodes are queued by the primary nodes in two separate queues: a queue for acquire requests and one for release requests. There is a dedicate message queue for each group of object identifiers, as depicted in Figure 51.

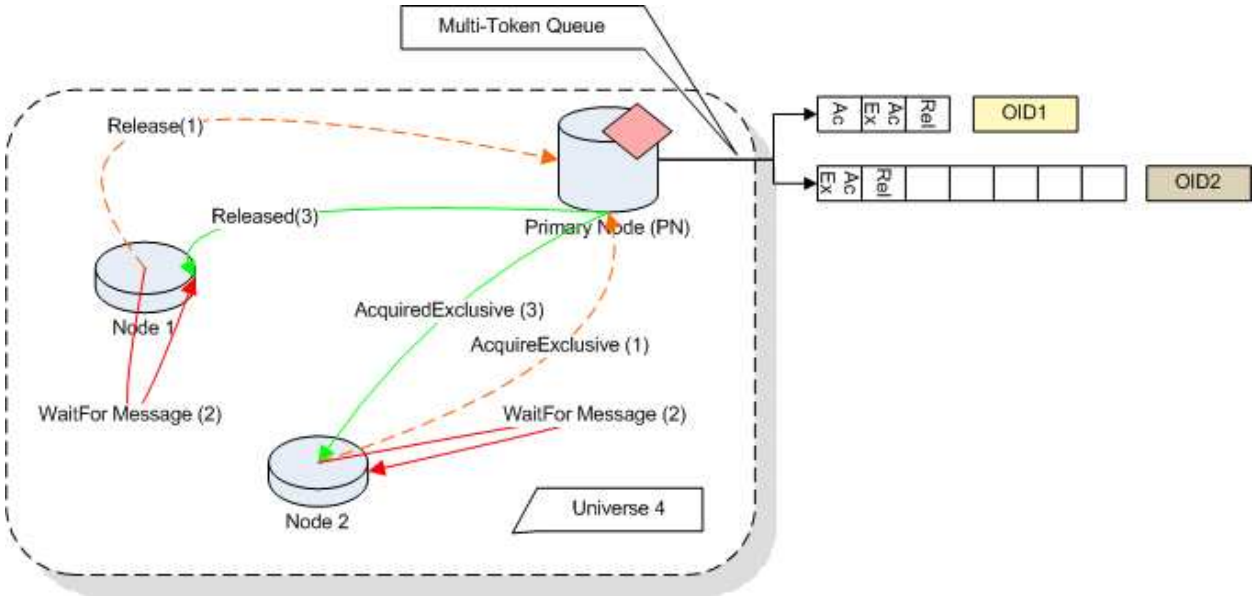

Figure 51: GUN Architecture: Message Queues

There are several worker threads that are processing the queued requests. In order to facilitate a higher parallelism level as well as lower locking time, the GUN prototype makes use of several worker threads that are handling the following operations:

- Acquire requests – serving acquire requests
- Acquire exclusive requests – serving acquire exclusive requests
- Release requests – serving release requests
- Token reception – serving token reception
- Latest nodes exchange – handling the set of latest nodes which hold the most up to date replica

Some of the kernel threads are depicted in Figure 52. In addition to the threads shown in Figure 52, there are a few more threads that are temporary used in order to perform certain cleanup operations.

Once the requests are queued on the primary nodes, they are processed one by one by the worker threads. In addition, the primary nodes are processing requests from other primary nodes that are received over the remote interface. These two kinds of requests are competing requests that are synchronized in the processing methods of the primary nodes using critical sections. It is worth to note

that all remote requests (e.g. remote from other universes via primary nodes) are received in the context of the RMI constructed thread. This means that the arbitration between local or remote requests is non-deterministic. However, in case of acquire-exclusive requests, GUN implements a priority mechanism, as described in [131], where requests coming from the local universe are handled before requests issued from other universes. The priority handling is time-based, meaning that requests are prioritized within a certain time-frame.

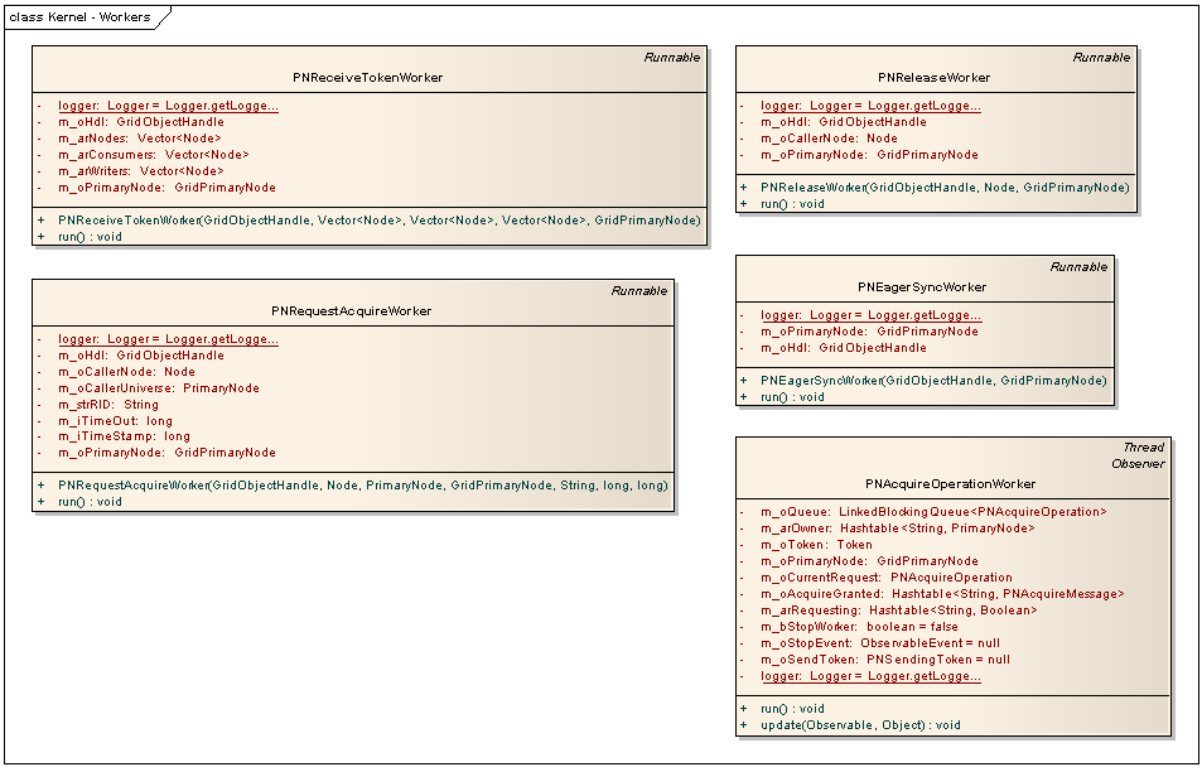

Figure 52: GUN Architecture: Kernel Worker Threads

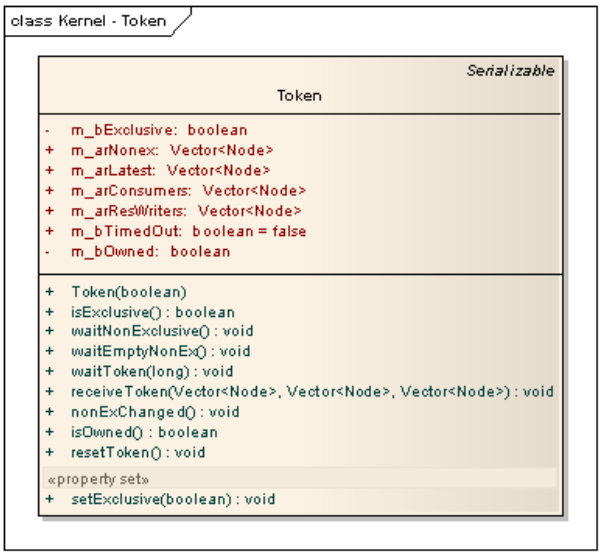

Figure 53: GUN Architecture: Kernel Token

The distributed mutual exclusion algorithm is based on the multi-token concept. For every group of object group there is a token associated. The tokens as well as all data structures are hashed based on the object identifier *OID.* The token structure is depicted in Figure 53. The token contains a list of nodes that are having requested the object in non-exclusive mode and did not release the objects yet. Second, the token contains a list of nodes that are holding an up-to-date version of the object. For specialized objects the token structure has been extended with a list of consumers and writers (for producer-consumer and result objects).

7.1.4 GUN Monitoring and Replication

In order to collect performance related data, a monitoring layer has been integrated into the grid primary node and grid node. The grid universe monitor which keeps track of the time spent for a given operation such as acquire time, acquire hits and misses and computes statistical information like acquire success rates. The monitoring components are invoked by the kernel in certain key points in order to log the required data. Performance data can be dumped into comma separated value files by invoking a method of the node where the client application is running. The monitoring classes are reflected in Figure 54.

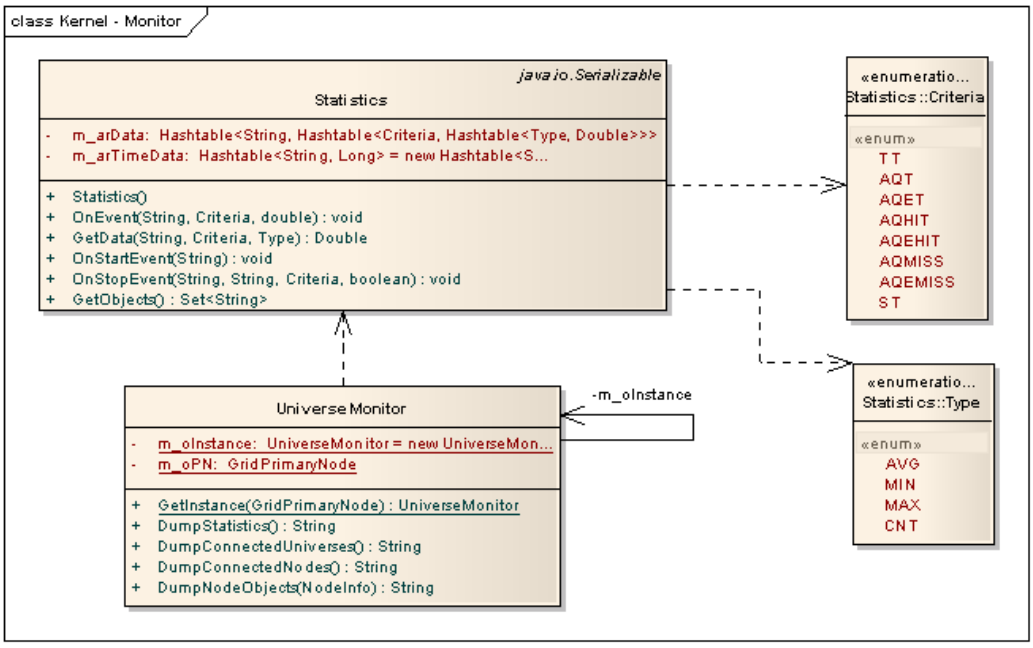

Figure 54: GUN Architecture: Kernel Monitor

GUN defines a generic replication hook that is called by the kernel when replication can be triggered. GUN contains a replication layer that takes care of object replication and migration, by applying a set of extendable dynamic replication rules that are supplied to the system at deployment time. Object replication and migration can happen either when an object is looked-up as there could be a closer replica, or during object acquiring and release. The replication mechanism is based on replication rules that are defined at deployment time and are loaded into the replication engine when the GUN system is started. If the replication engine decides to replicate a given object, the object is replicated to the

designated target node and the client handles are updated so they refer to the newly created replica. The replication engine and the replication hook that is called by the GUN kernel are shown in Figure 55.

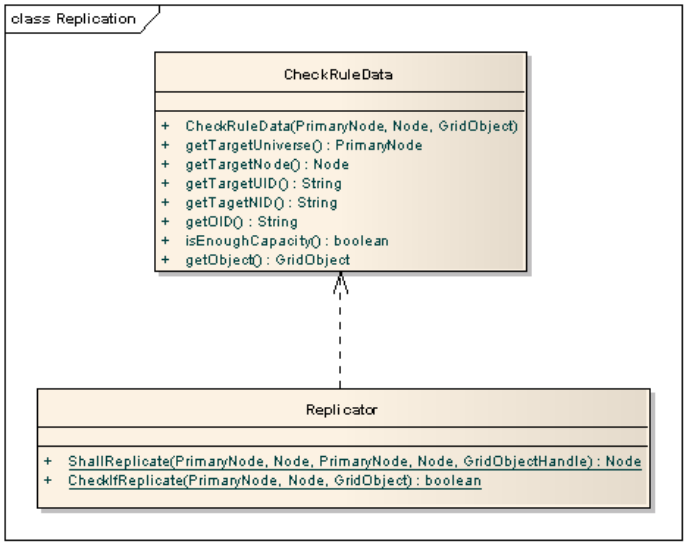

Figure 55: GUN Architecture: Replication Engine and Replication Hook

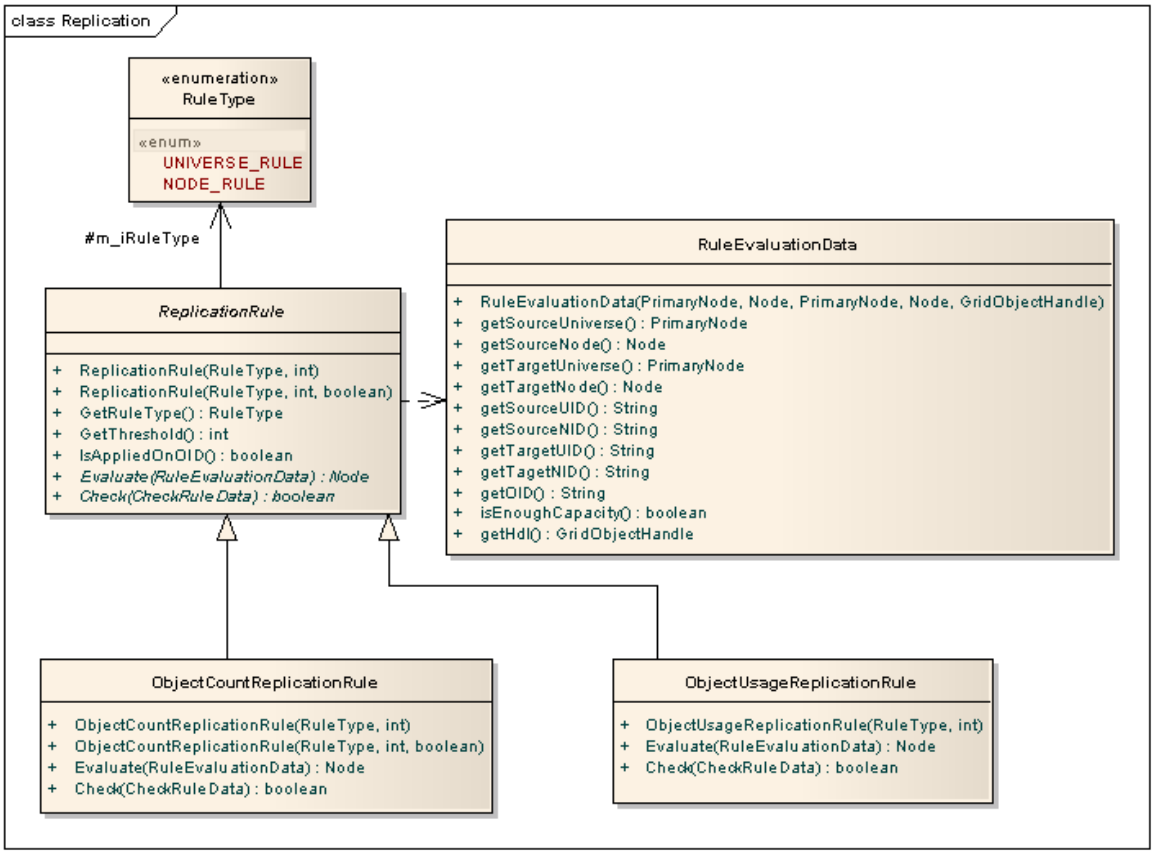

Figure 56: GUN Architecture: Replication Rules

The basic replication rule classes are presented in Figure 56. It is possible to define custom replication rules and register them into the replication engine. The rule evaluation is left to the concrete

rule implementation. The replication engine simply collects the evaluation of all replication rules. If one of the rules evaluates to true, the replication engine triggers object replication.

7.2 Experimental Results

The GUN prototype was tested in a large scale grid environment which is depicted in Figure 57. During the prototype development and testing phases, a number of 3 to 5 universes were deployed. The first universe is represented by a small cluster physically deployed at the “Politehnica” University of Timisoara which is part of the MedioGrid project [134]. The second universe is made out of machines of a high performance cluster located at the Western University of Timisoara which is used in the SCIEnce European project. The third universe contains machines located at the Research Institute for Symbolic Computation (RISC) in Linz, Austria. The last two universes consist of machines located in a cluster of the “Technische Universität Dresden”, Germany and the Bangalore Institute of Technology University, India. In this configuration, the latencies between universes were ranging between 10ms and 240ms. The very first experimental results [133] have confirmed the initial hypothesis of the feasibility of shared data programming approaches for large scale distributed systems.

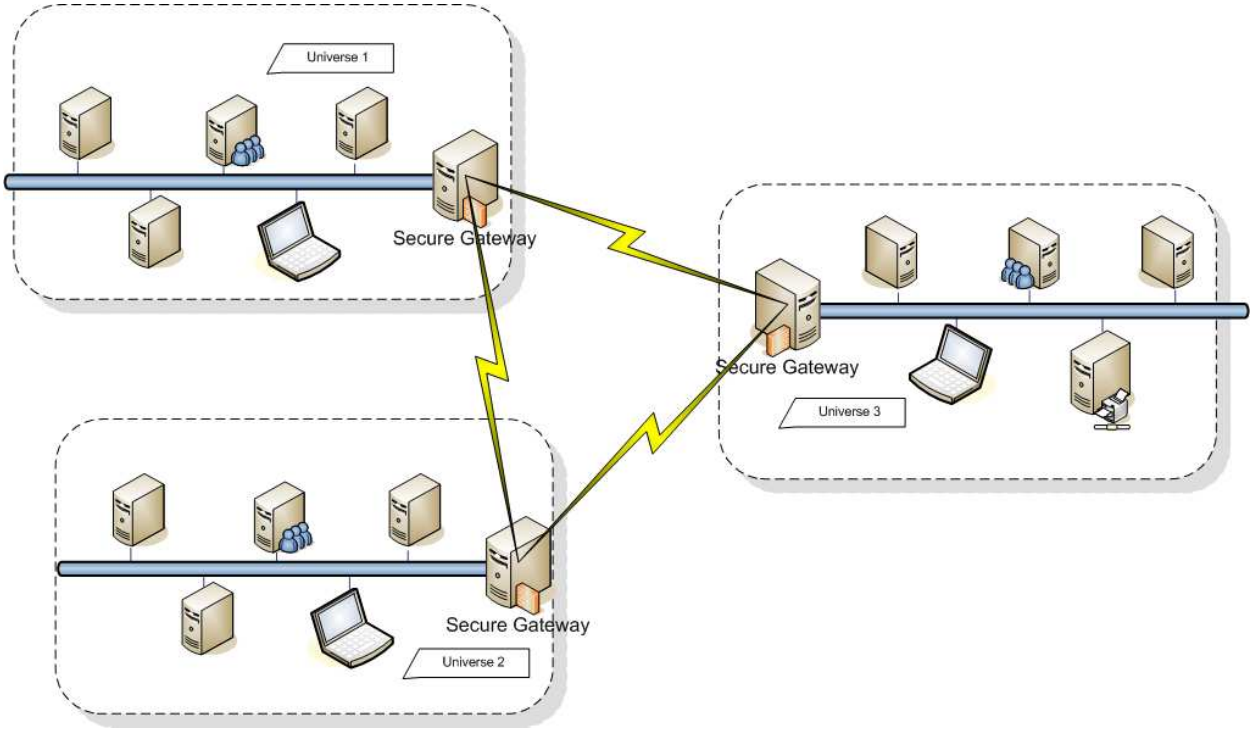

Figure 57: GUN Experiments Setup

One of the first problems encountered during the execution of the first experiments was that the number of required nodes was higher than the number of available machines. Most of the clusters that were accessed had between 3 and 10 machines and the experiments require a number of nodes ranging from 5 to 50 (in the ideal case). The problem was exacerbated because there were no common access interfaces to those networks so that a significant effort was required to deploy GUN and start the experimental applications. As a result, a better experimental setup was necessary.

An SGI Altix 4700 machine was used, which is available at the Research Institute for Symbolic Computation (RISC) in Linz, Austria and at “Technische Universität Dresden”, Germany. The machine located at RISC has 128 Intel Itanium 2 Montecito processors with hyper-threading technology, running at 1,6GHz and having 18 MB L3 cache which means that it can execute 256 threads simultaneously. The machine has 1 Terabytes of main memory (Global Shared Memory NUMA) and 24300 GB data storage. The machine located at TU Dresden's Center for Information Services and High-Performance Computing (ZIH) is powered by 2048 Intel Itanium 2 Montecito processor and has 6.5 Terabytes of main memory. This machine can deliver 11.9 trillion floating point operations per second (TFLOPS), making it the most powerful system in the East German Federal States.

The second problem appeared due to the Java runtime environment which was provided by the JRockit java distribution which is supposed to be optimized for the Altix machines and the Linux SUSE operating system. Preliminary tests showed a non-scalable behavior of the GUN prototype although the theoretical analysis highlighted the contrary situation. Investigations that were done during almost one month time showed that the JRockit runtime environment is not scalable in terms of the RMI server implementation. In other words, the RMI server does not scale to a high number of clients and therefore introduced a bottleneck in our system. The problem was fixed by using an adapted version of SUN’s JDK version 1.6 for 64 bit machines which does not have the same scalability problem.

The environment problems were not completely overcome because the latency between universes was not present anymore in a natural way. As a result, artificial delays between remote calls were introduced in the GUN prototype in order to reflect a real deployment. In order to reduce the risk of uncontrolled thread scheduling, a spinning wait was used. A latency of 10ms was considered for calls within a universe and 50ms for calls from one universe to another. It is important to note that these values are highly dependent on the remote method signature as well as their values (e.g. in case of a list of various objects), as all method parameters are serialized and transferred over the network. This aspect was not addressed in the experiments running on Altix as it naturally happens in a real deployment, because in case of GUN, there is no significant data marshaling between remote machines. However, the remote execution penalty in the real wide scale distributed environment was not higher than 250ms, considering the parameters for all remote methods defined in GUN. In case of only the European clusters, the latency was between 40ms and 90ms. As a result, the fixed value of 50ms was considered in the Altix evaluation setup.

The bulk of the both qualitative and quantitative results have been gathered during the execution of the prototype experiments, which took more than 3 months to complete. Some experiments required 5 to 6 hours to complete and some of them needed to be run again to match the same overall system load (as the experiments have been conducted on a real system where a large group of scientists were working on). In several cases multiple runs were necessary in order to rule out extreme situations (e.g. high system load) and have a more balanced measured data set.

7.2.1 Grid Object Search

The experiments for grid object search that are described in Chapter 6 specified a number of nodes up to 50 nodes per universe and 1000 grid objects. Due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

In case that “one object per node” replication rule is used, the search time dependency to the number of objects, ranging from 1 to 1000 is shown in Figure 58 for a configuration of 30 nodes per universe. The first group of 30 client nodes is the one belonging to the universe where the initial object has been created. Due to the replication policy, an object replica exist in all nodes of every universe, thus the same search time is experienced by all applications independent on their universe membership.

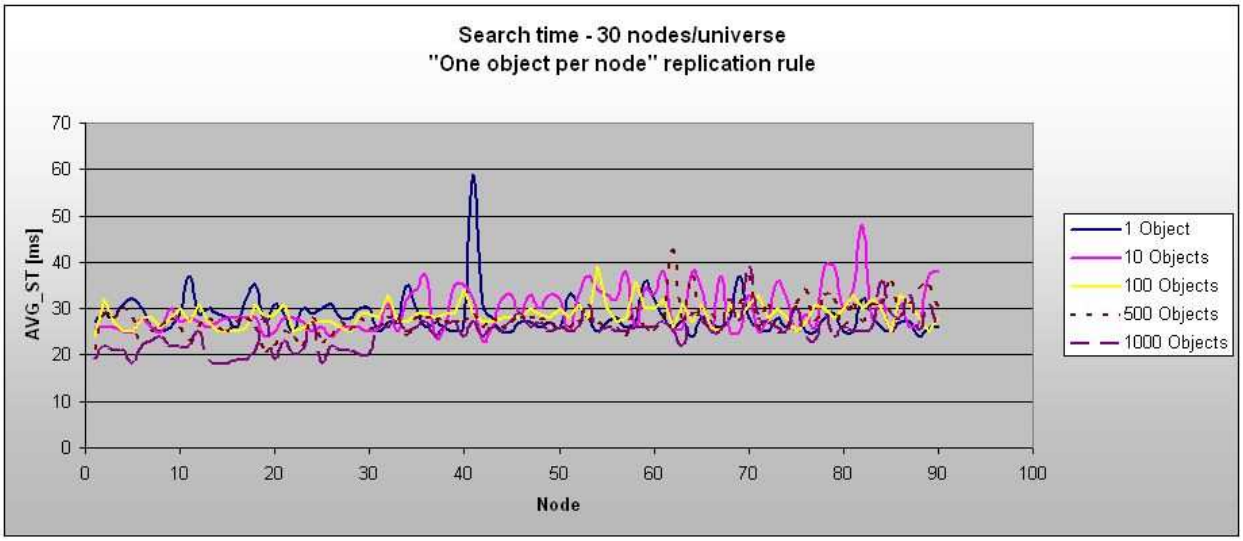

Figure 58: Grid Object Search – One object per node

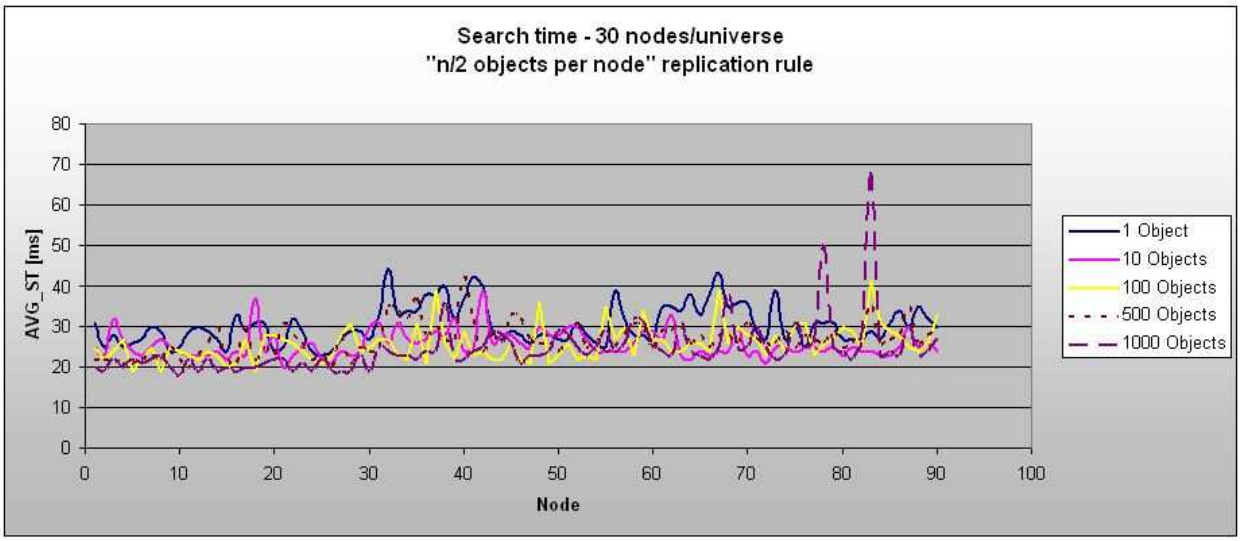

Figure 59: Grid Object Search – n/2 objects per node

Similarly, the same results can be observed in Figure 59 and Figure 60 where “*n/2* object replication” and “one object per universe” replication rules are used. In all previous situations we observed a uniform distribution of search time values across universe nodes, independent to the number of grid objects. Some of the occasional search time oscillations can be accounted on system load burst as well as unpredictable operations of the java run-time environment (e.g. garbage collector routine execution).

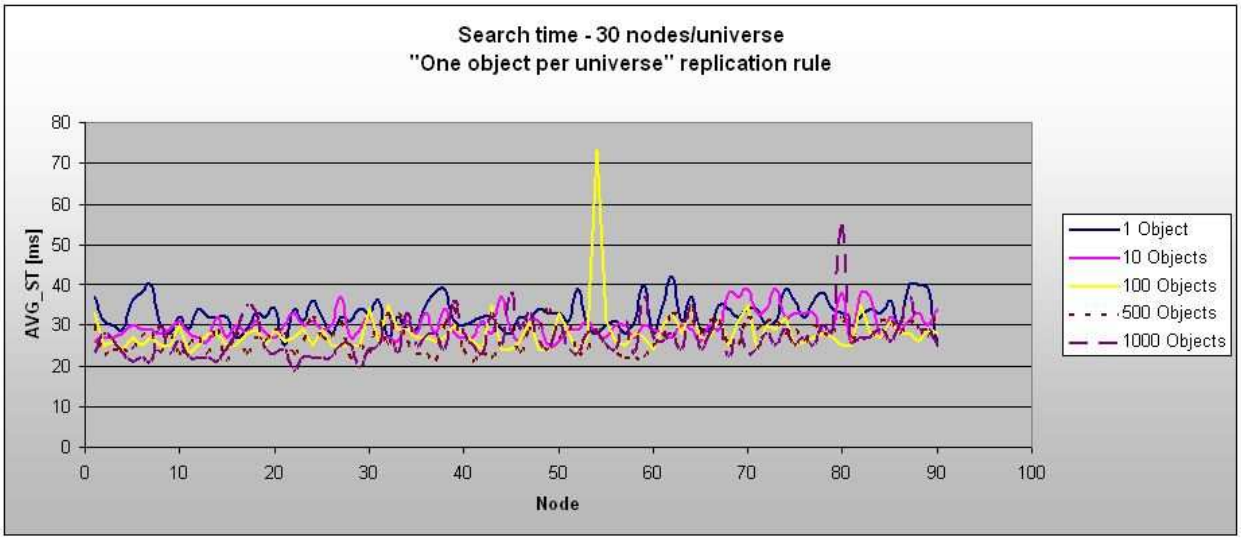

Figure 60: Grid Object Search - One object per universe

In case when no replication is used, the search time dependency to the number of objects, ranging from 1 to 1000 is shown in Figure 61 for a configuration of 30 nodes per universe. The first group of 30 client nodes belongs to the universe where the object was created. As a result, the clients residing in that universe have small search time values. The next group of 30 clients is located in a different universe and the search request if forwarded across the universe registration graph. In this case the first contacted universe during the search operation has the object. As a consequence, the search times are higher with approximately 100ms (round trip request). The last group of 30 nodes belongs to a different universe. The search requests are first directed to the second universe's primary node and then to the first universe where the object resides. Thus, the search time is approximately twice the value as in the later case.

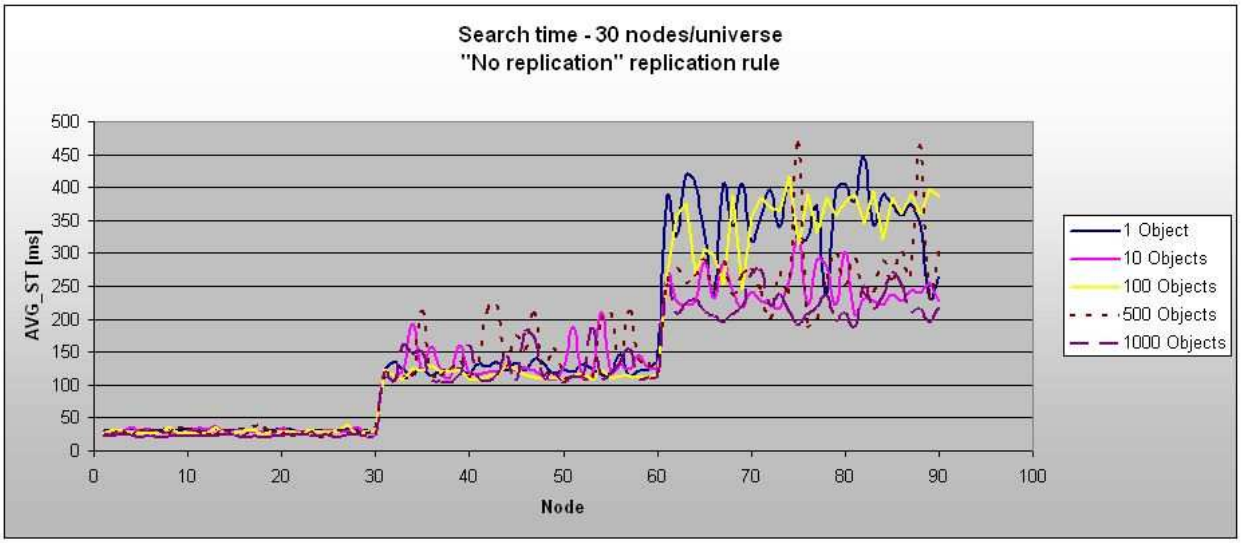

Figure 61: Grid Object Search – No replication

The experiment results with different number of nodes within a universe, in case of 1000 grid objects where one object per node and no replication is used are presented in Figure 62 and Figure 63. It can be noticed that the search time does not depend on the number of nodes. In case where no replication is used, the same pattern described in Figure 61 occurs: the search time increases together with the search chain length. For every search indirection, the search time increases by one round-trip inter-universe latency (approximately 100ms).

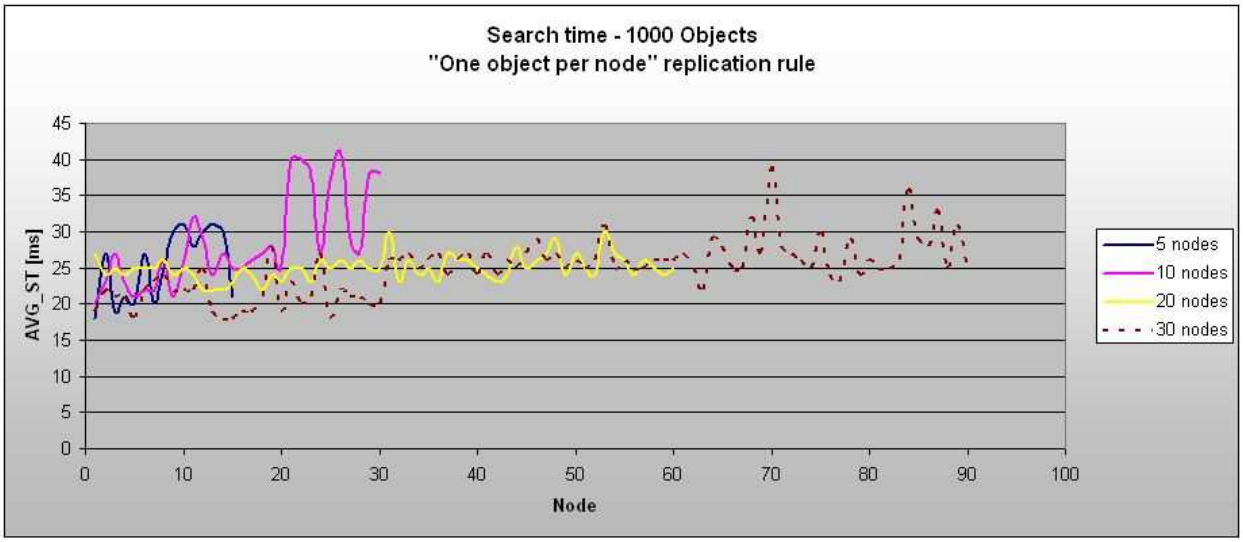

Figure 62: Grid Object Search – Node dependencies with replication

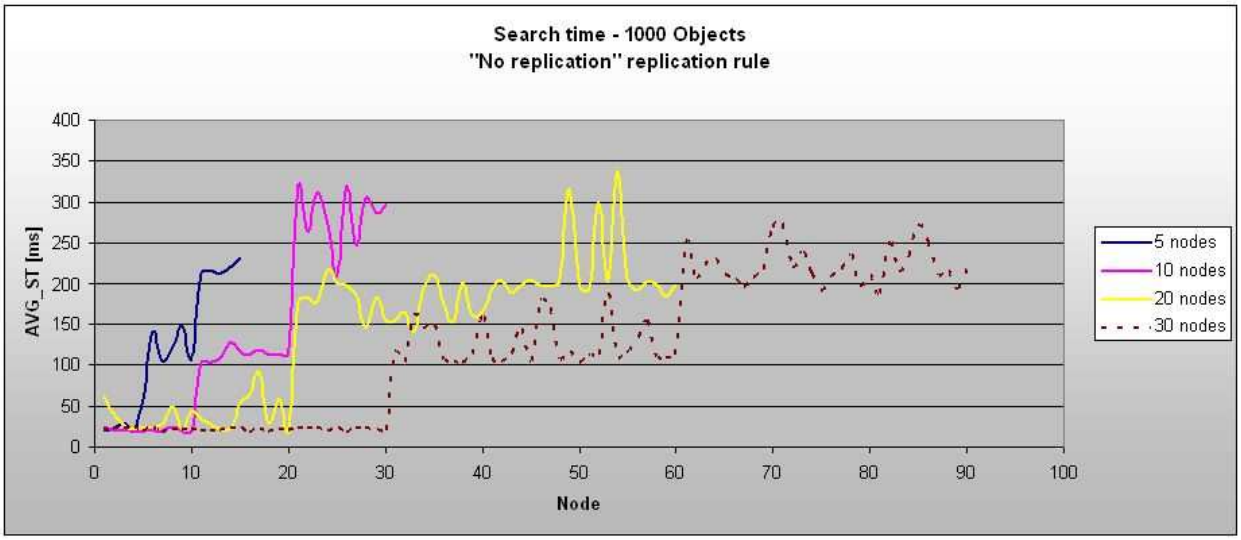

Figure 63: Grid Object Search – Node dependencies without replication

7.2.2 Acquire Correctness

The experiments for the acquire operation that are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe (one node runs on one core) and a number of 3 universes.

During all the experiments we have noticed a 100% success rate for all acquire operations (AQHIT=1) that were issued, considering a timeout value of 2000ms. First, in case that only one client was running and the token was held in a remote universe, in a 30 node per universe configuration scenario, the results summarized in Table 12 were obtained. Since there is only one worker and the number of requests are constant over a period of time, there are no variations of the acquire time depending on the acquire request frequency. These values are reference values that apply to a relaxed system with seldom operations.

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
OST [ms]	33	31	31	21	30
AQT [ms]	105	116	106	109	110
AQHIT	100%	100%	100%	100%	100%
AQMISS	0	0	0	0	0
RLT [ms]	37	41	44	40	44
TRAQ	0.008	0.007	0.007	0.008	0.008

Table 12: Acquire measurements for one worker

In the same deployment of 30 nodes per universe, but where a client is running on every node, depending on the delay between subsequent operations the results illustrated in Figure 64 were obtained. The first group of 30 nodes belongs to the universe where the token resides. As there is no acquire exclusive issued in this scenario, the token remains fixed, thus the nodes within the same universe experience a very low acquire time. The other nodes which belong to universes which do not hold the token experience increasing acquire time values as the delay between subsequent requests decreases.

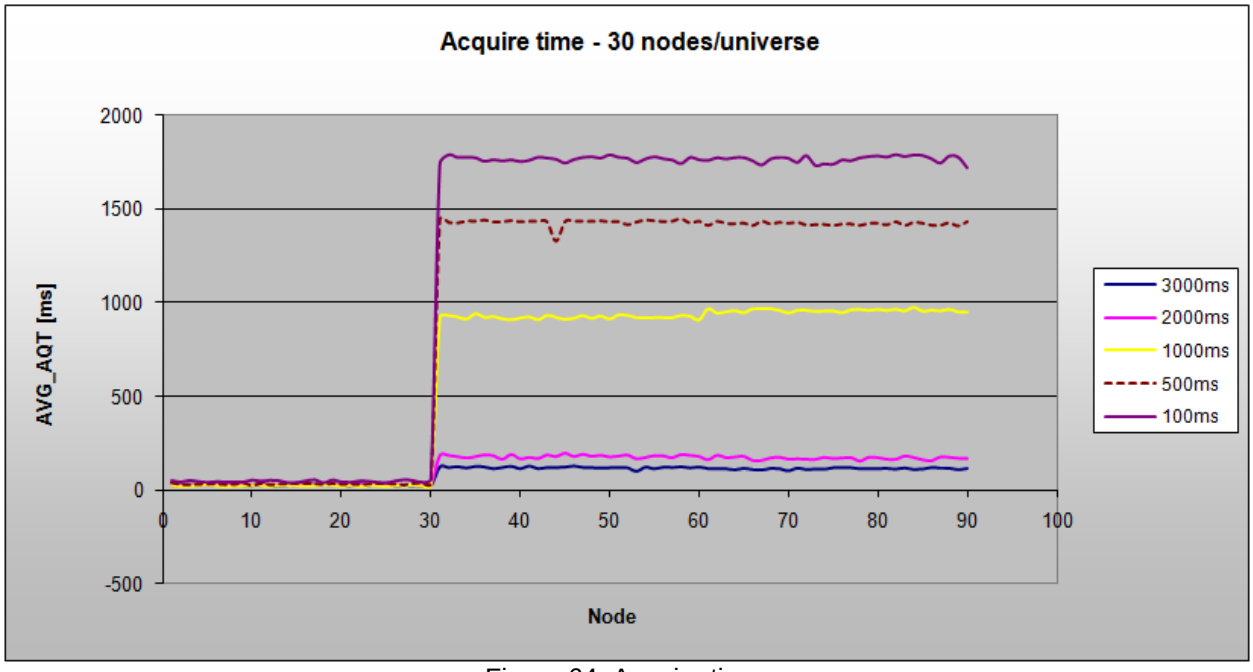

Figure 64: Acquire time

As seen in the diagrams, in case the acquire operations are issued with a delay of 2000ms respectively 3000ms, GUN shows a good and stable performance independent on the node location.

There is only a slight increase in acquire time for the nodes belonging to the universes that do not have the token. This is quite normal since the acquire request has to pass the universe boundaries (remote call over large latency connection). In the other three cases, there is a performance degradation in the system when the acquire requests are issued more rapidly. This happens because all nodes are issuing request towards the primary node that holds the token and the requests are serialized in a queue. If the request frequency is higher than the processing frequency, the requests are accumulating in the queue and the waiting time increases (see Table 12). It is worth to note that the processing frequency depends on the inter-universe network latency because the responses are sent via a large latency connection.

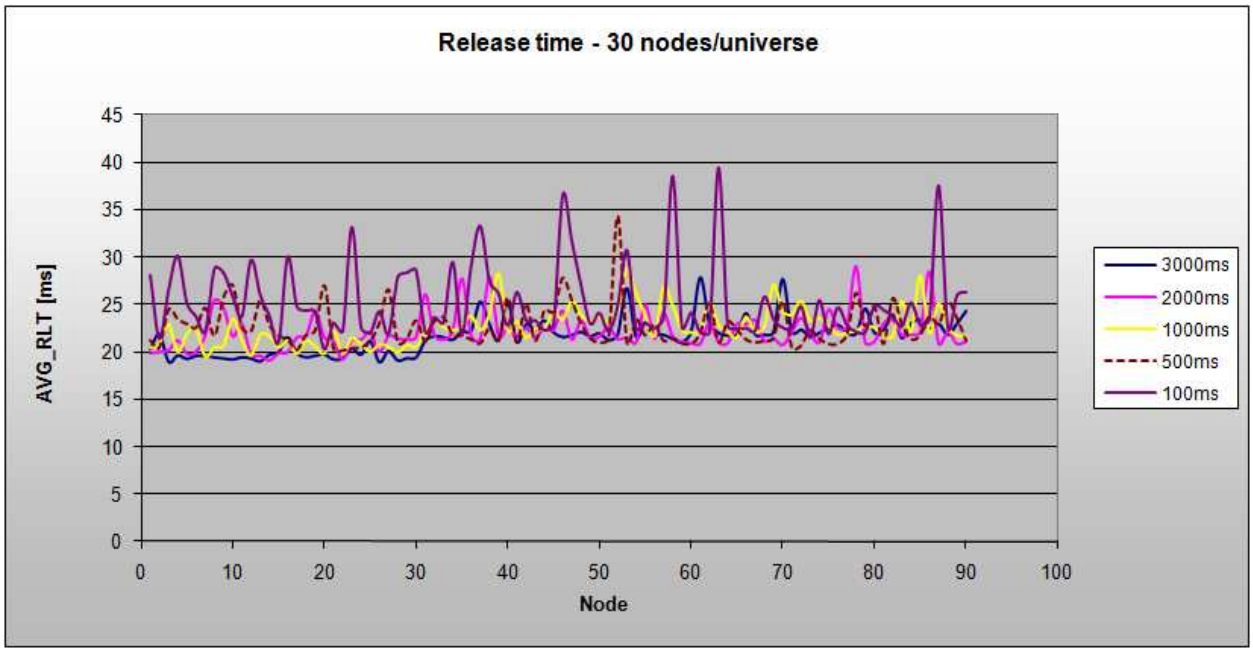

Figure 65: Release time

As shown in Figure 65, contrary to the acquire time, the release time remains stable independent on the request frequency (acquire frequency is equal to the release frequency) since the release operation is implemented asynchronously and a response is not awaited from the primary nodes. Only in the extreme case where requests are issued within each 100ms, a variation of the release time can be noticed but this can be accounted to the global task scheduling mechanism. Completion time shown in Figure 66 highlights a shape resembling the acquire time.

The performance results for average acquire time if different nodes and request frequencies were used are shown in Figure 67, Figure 68, Figure 69 and Figure 70. It can be observed that the same pattern described above appears in all node configuration with the remark that the impact on the acquire time is direct proportional to the number of nodes. This aspect is obvious since the number of requests is dependent on the number of nodes.

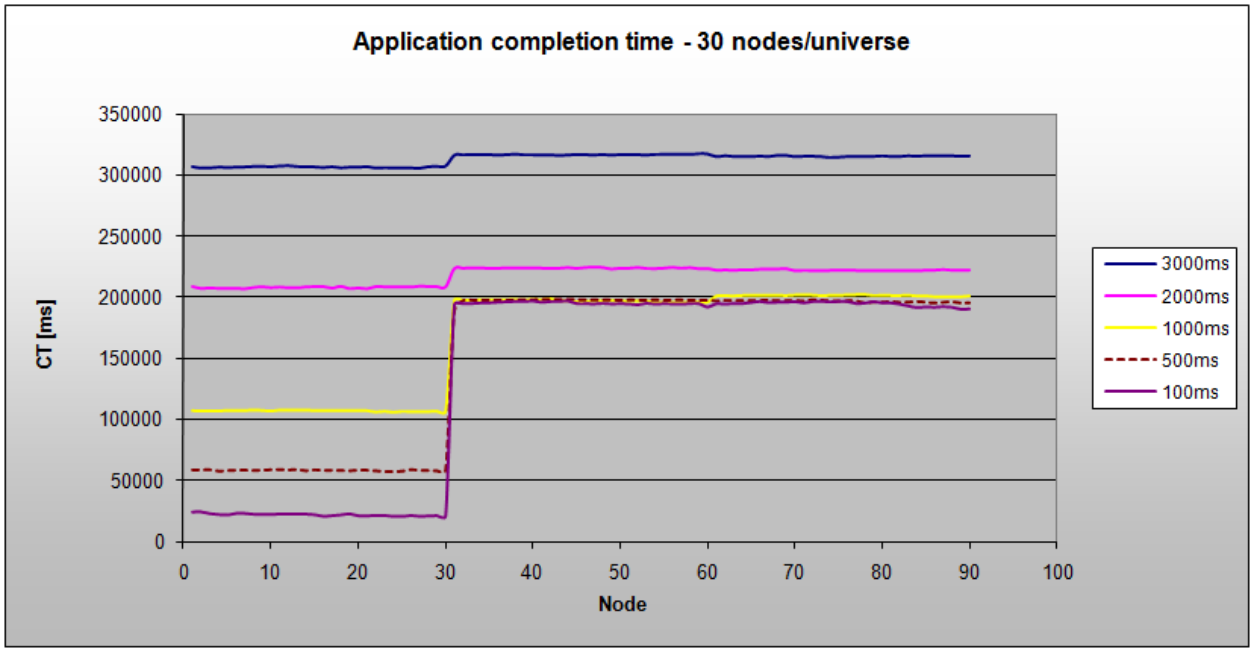

Figure 66: Completion time

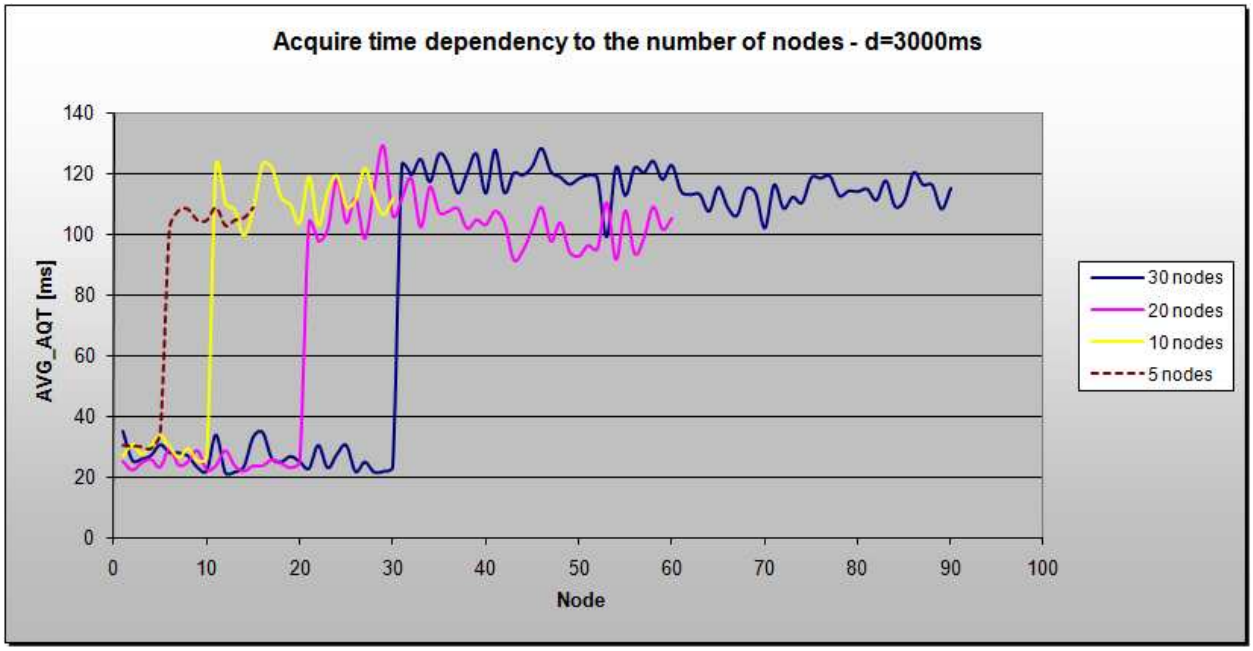

Figure 67: Acquire time dependency to the number of nodes, d=3000ms

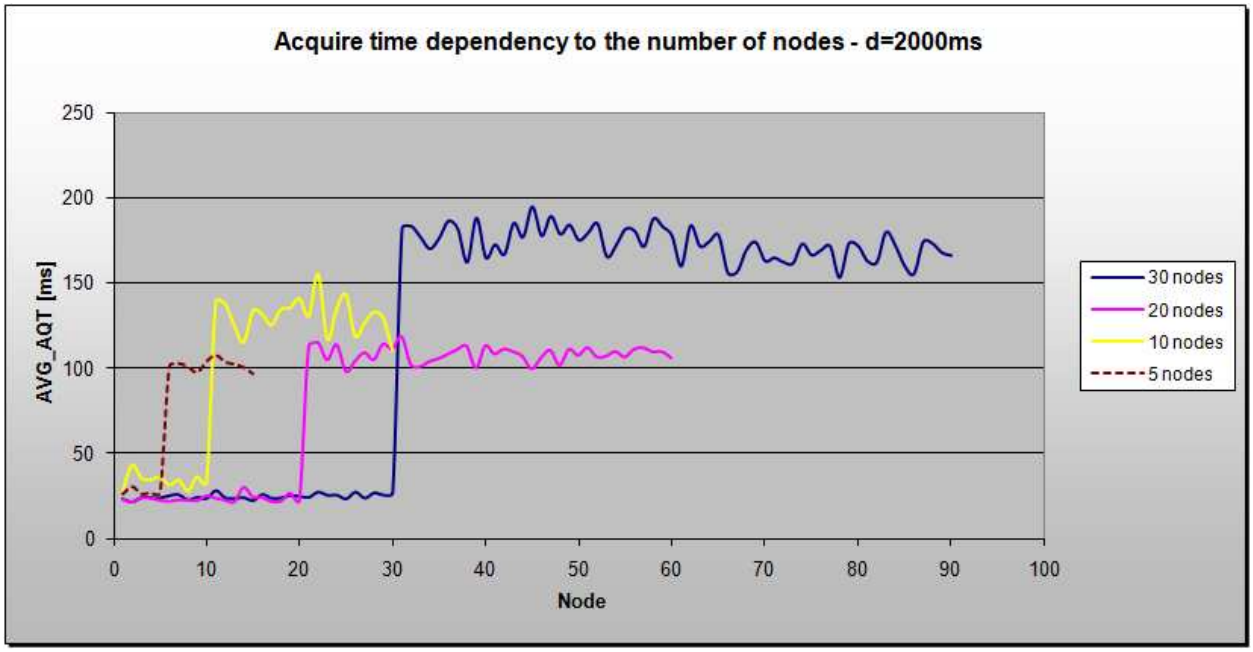

Figure 68: Acquire time dependency to the number of nodes, d= 2000ms

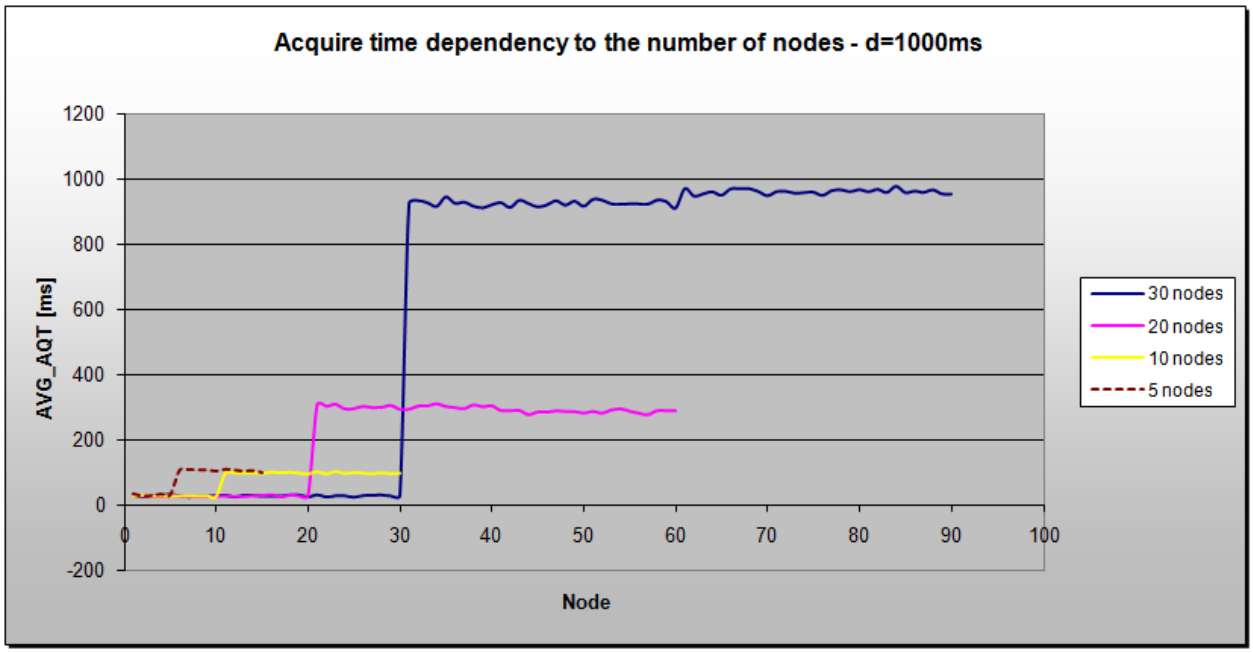

Figure 69: Acquire time dependency to the number of nodes, d=1000ms

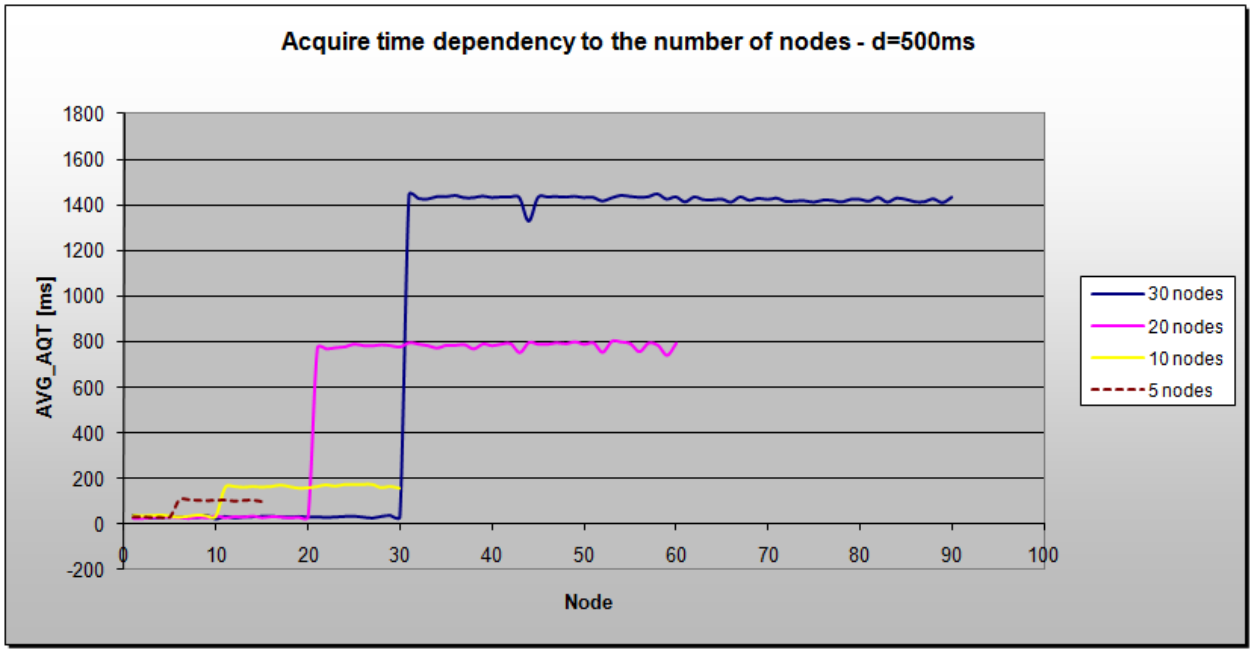

Figure 70: Acquire time dependency to the number of nodes

In case of the release operation, there is no dependency to the number of nodes, as shown in Figure 71.

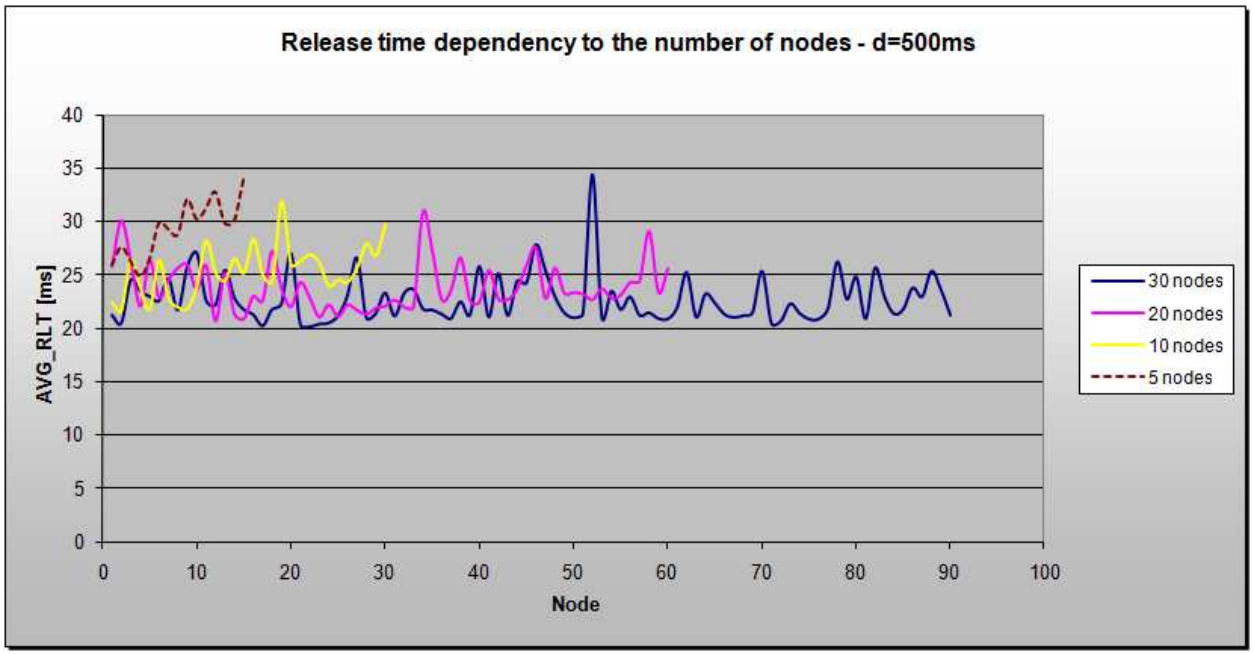

Figure 71: Release time dependency to the number of nodes

7.2.3 Acquire Exclusive Correctness

The experiments for acquire exclusive operation that are described in Section 6.2.3 specified a number of nodes up to 50 nodes per universe. Due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3

universes. The timeout value was set to 2000ms and during all the experiments a 100% success rate was obtained (AQHIT=1).

The correctness of the GUN implementation for acquire exclusive was assessed according to the specifications of the entry consistency model. In our experiment, each grid shared data contains a counter which is incremented each time the value of the object is changed. Since the order of acquire operations is the same from every node's point of view, the interleaved sequence of object counters must be ordered. Such an ordered sequence of object versions was witnessed for all the run experiments.

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms	d=5000ms
TT [ms]	83	74	79	80	80	78
MAX(OST) [ms]	134	145	122	134	155	169
AQET [ms]	98	88	102	104	98	110
AQEHIT	100%	100%	100%	100%	100%	100%
AQEMISS	0	0	0	0	0	0
RLT [ms]	32	34	25	26	25	26
TRAEQ	0.008	0.008	0.008	0.008	0.008	0.009

Table 13: Acquire exclusive measurements for one client

In case there is only one client issuing acquire exclusive requests, and the token resides in another universe, the measurements shown in Table 13 were obtained for different values of the delay *d*. In the table the maximum value of OST is considered since the objected is synchronized only once (only one client modifies the replica). Subsequent acquire exclusive requests will not determine any object synchronization anymore, thus OST reduces to a local function call (~10ms) and thus the average OST value tends to become very small.

In case of 30 nodes per universe deployment where a client is running on every node and is issuing acquire exclusive requests as specified in the experiment description, the average acquire exclusive time for 7 different values of the delay parameter is represented in Figure 72. If the delay is higher than 1000ms, acquire exclusive time does not show any significant fluctuations between universe nodes and remains stable between 400 and 1000ms. If the delay is reduced below 1000ms, acquire exclusive time increases. The explanation is that smaller delays lead to a higher number of requests per time unit. As the processing capability of the primary nodes has an upper bound, requests accumulate in the primary node's queues and thus increasing the processing time.

In case of the same deployment, release time is represented in Figure 73. Similar to acquire operation, the release operation corresponding to acquire exclusive requests shows a constant time independent on the number of nodes or the delay between subsequent requests. Same as for acquire operation, the release operation corresponding to acquire exclusive requests uses an asynchronous mechanism where the release of the shared object is not awaited anymore. As a consequence, every acquire operation might be blocked until all pending releases are performed. This solution increases the system's throughput and increases the chances of higher concurrency.

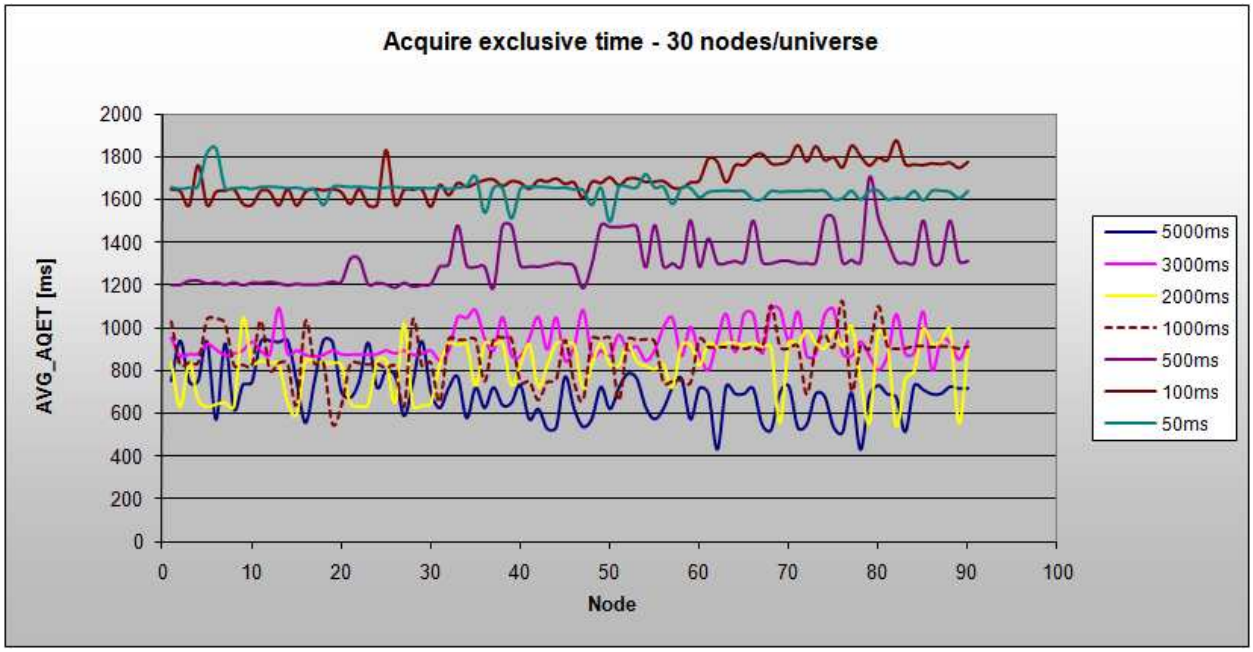

Figure 72: Acquire exclusive time

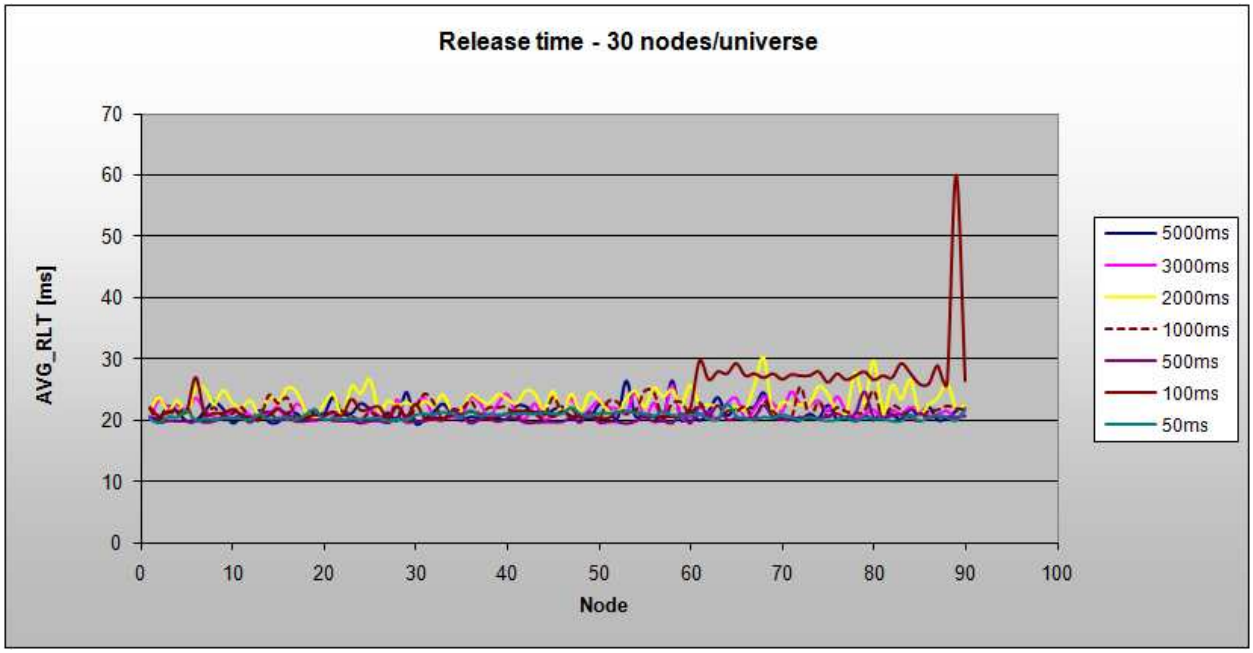

Figure 73: Release time

The performance of acquire exclusive operation in respect of the number of nodes for *d*=5000ms, *d*=2000ms and *d*=50ms is illustrated in Figure 74, Figure 75 and Figure 76. As expected from the previous results, if the delay between operation is large enough (e.g. 5000ms) the number of nodes does not have an impact on acquire exclusive time. However, if the delay is smaller (e.g. 2000ms) there is already a noticeable difference as seen in Figure 75 or Figure 76. As the number of nodes increases, the number of issued requests per time unit increases. This effect is similar to decreasing the time between operations which cause an increase of the number of issued requests per time unit too.

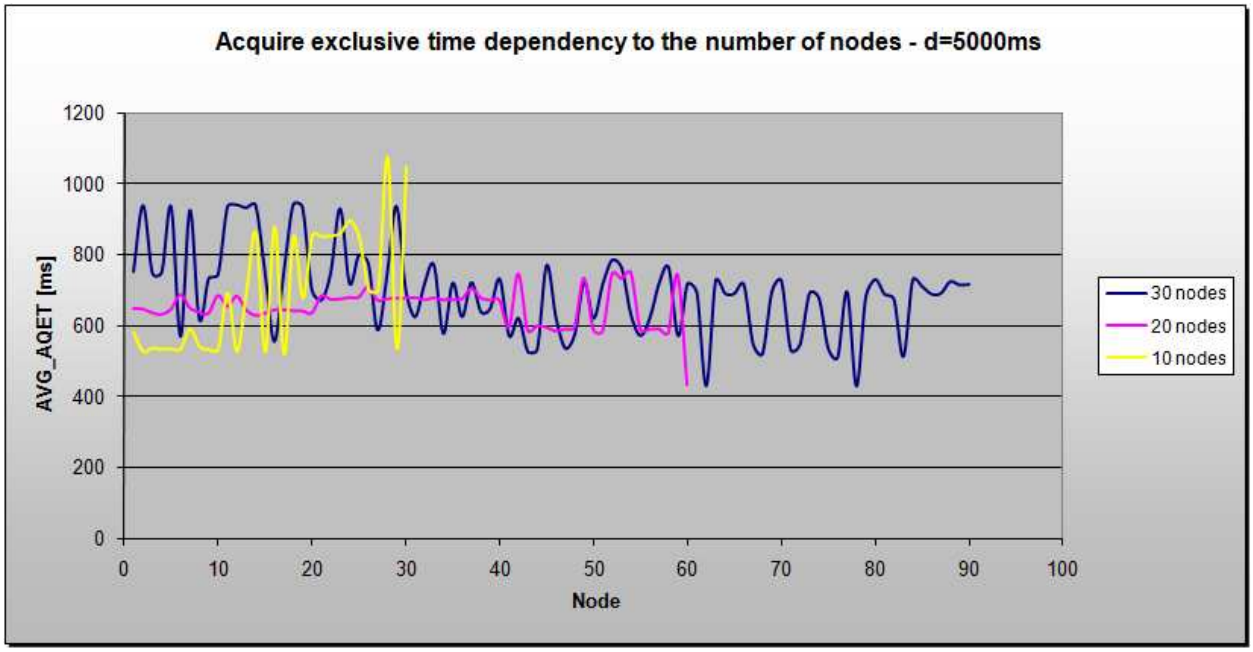

Figure 74: Acquire exclusive time dependency to the number of nodes, d=5000ms

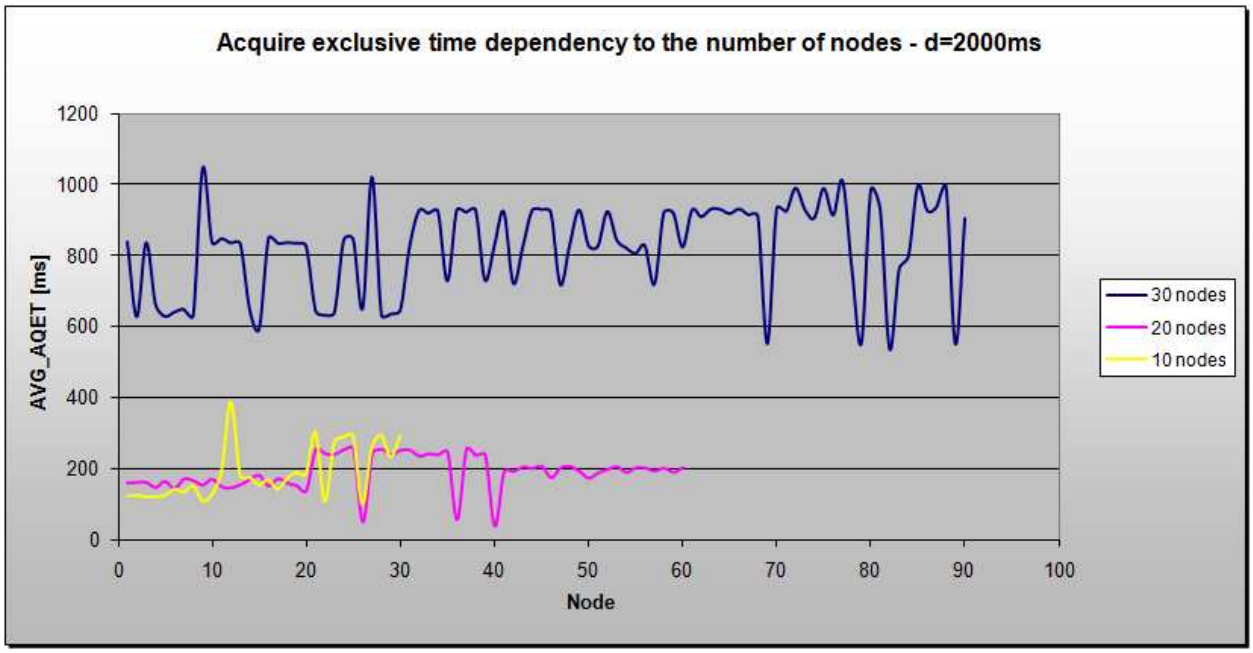

Figure 75: Acquire exclusive time dependency to the number of nodes, d=2000ms

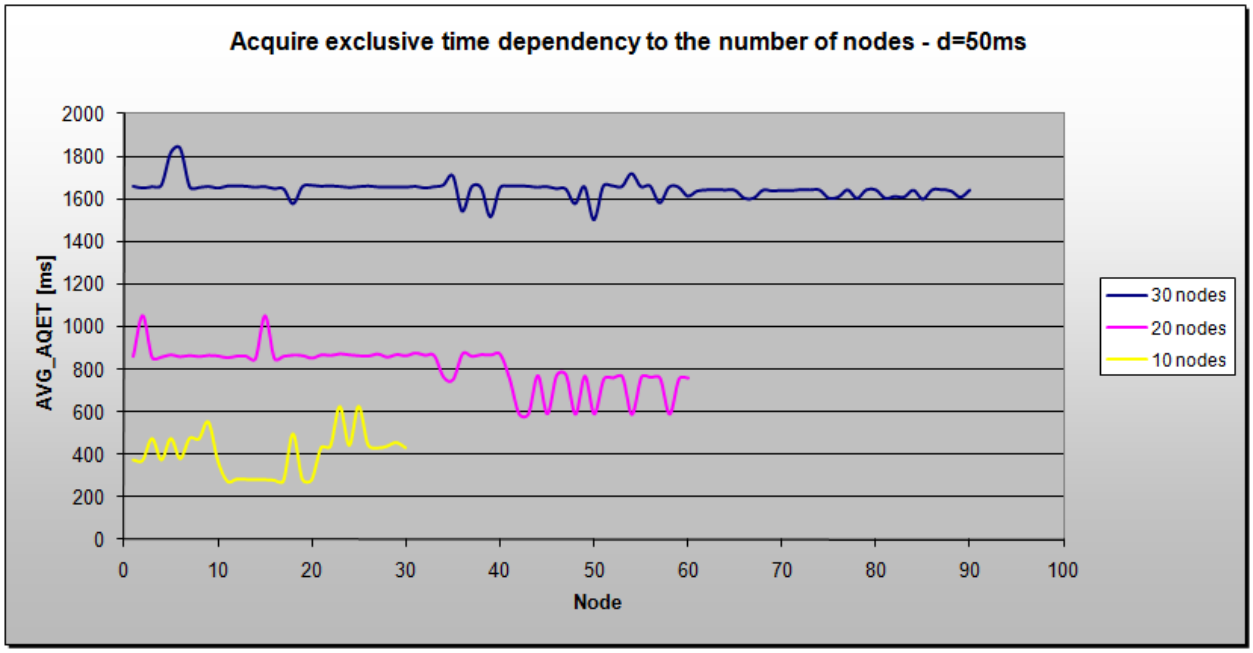

Figure 76: Acquire exclusive time dependency to the number of nodes, d=50ms

Due to the asynchronous implementation, as expected and represented in Figure 77, the release time is not dependent on the number of nodes. As explained above, potential stalls due to pending releases are only visible as increases of the acquire time for pending acquire operations.

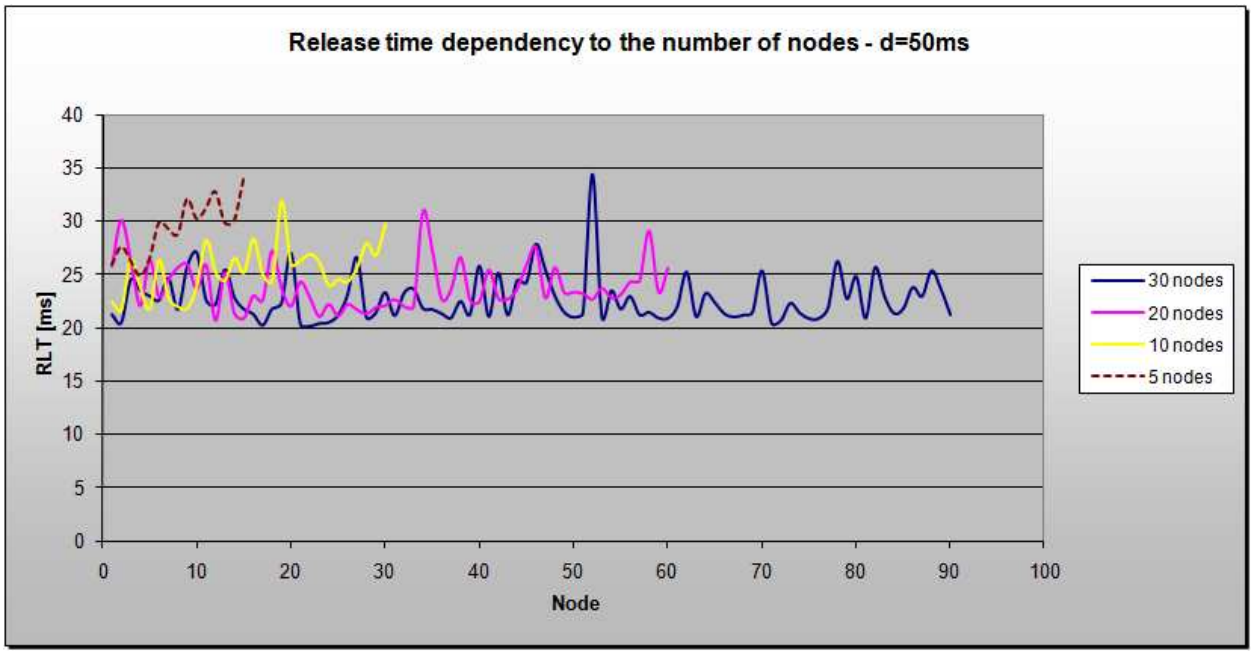

Figure 77: Release time dependency to the number of nodes

7.2.4 Grid Read-Only Objects

The experiments for read-only objects which are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Due to physical limitation in available computing cores the

experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

In case of read-only objects the acquire time is close to zero as no locking is necessary since the object's state is immutable. The same applies for the release operation. In this respect, the difference to the generic grid object is evident (assuming that the read only object is replicated to the caller node and thus no remote invocation cost is incurred). Since there is no logic behind the acquire operation there is no dependency to the number of nodes. Some of the measurements for the situation where there is only one client are summarized in Table 14.

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
AQT [ms]	0	0	0	0	0
AQHIT	1	1	1	1	1
AQMISS	0	0	0	0	0
RLT [ms]	0	0	0	0	0
CT [ms]	13142	53376	103184	203652	303187
TRAQ	0.064	0.054	0.059	0.052	0.063

Table 14: Read-only object measurements for one client

The application completion dependency to different values for the delay *d* is shown in Figure 78. Since acquire and release time is zero, the differences are accounted to the value of the delay multiplied by the number of runs. For example the difference between the first two situations where *d*=3000ms and *d*=2000ms is equal to (3000-2000) * 1000 = 100000ms which is the total delay time introduced in the experiment.

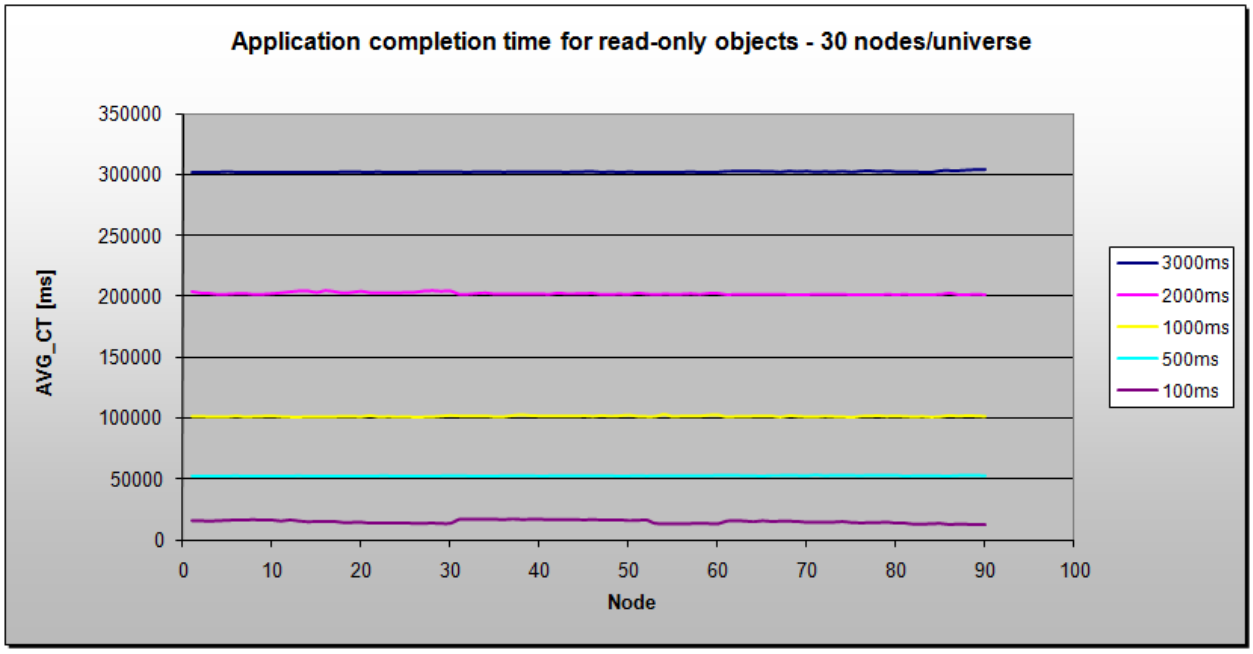

Figure 78: Completion time for read-only objects

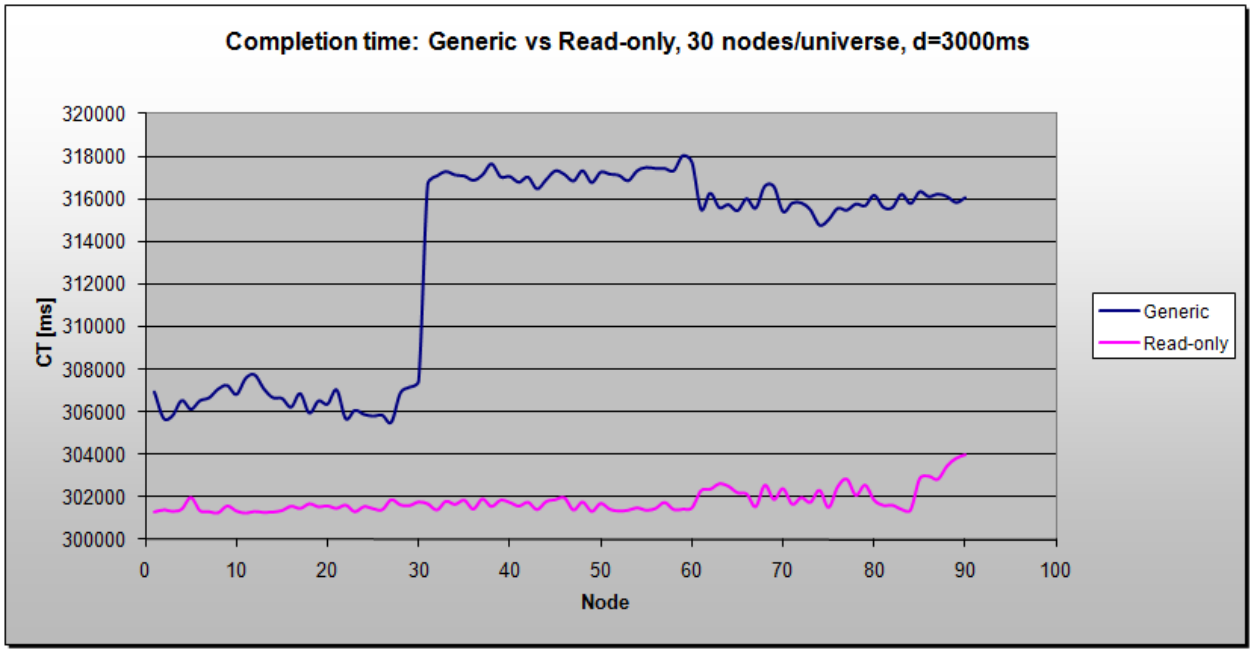

Figure 79: Completion time: read-only vs. generic objects, d=3000ms

A more interesting aspect is the difference in the application completion time between read-only and generic objects. The results for *d*=3000ms and *d*=100ms are shown in Figure 79 and Figure 80

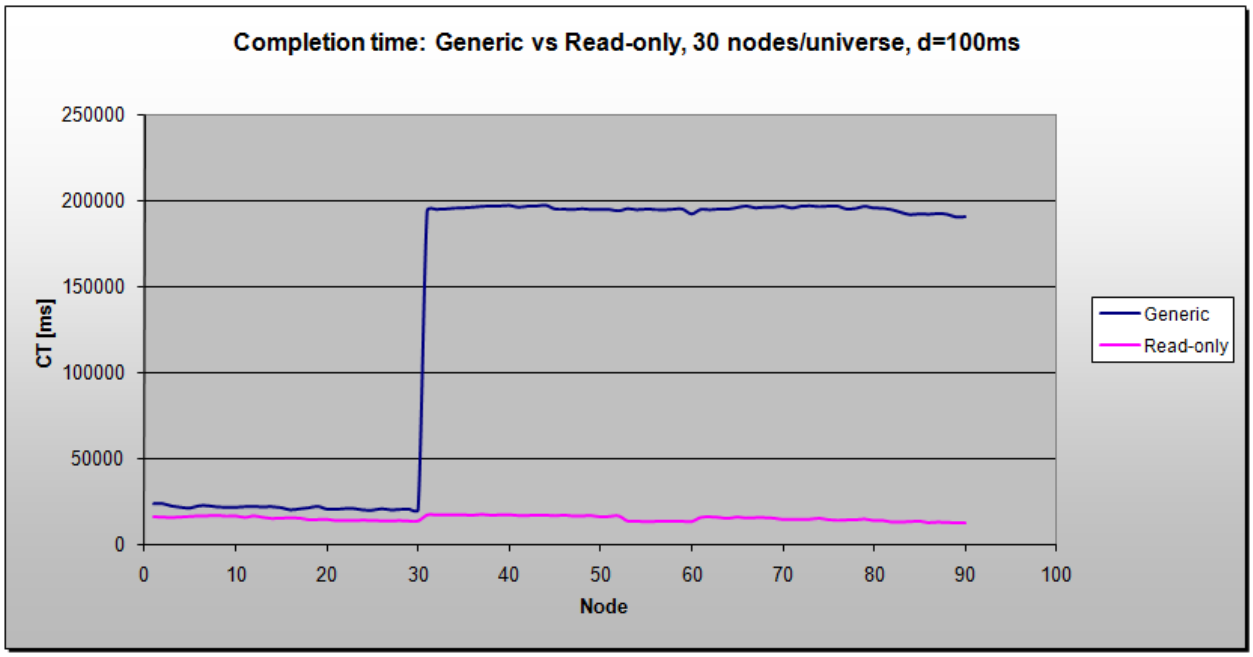

Figure 80: Completion time: read-only vs. generic objects, d=100ms

. In case of the generic objects, the first group of object resides in the universe containing the token and is experiencing significantly lower values for the completion time. The other group of objects require to contact the primary node containing the token in order to get the acquire request granted and thus have higher completion time values. As seen in both cases, there is a significant advantage of using read-only objects where frequent object state must be read especially when the location of the token is

uncertain (e.g. one client performs acquire exclusive operation and all other requests have to be redirected to a remote universe).

7.2.5 Grid Private Objects

The experiments for private objects which are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

The first part of the experiment refers to ***acquire operations***. In case of a 30 node deployment where only one client is issuing requests and the client is located on the same node as the shared objects, the measurements presented in Table 15 and Table 16 were obtained. As can be seen by comparing the two tables, the private object offers much better performance. Virtually the acquire time are insignificant, the cost of the operation is basically a local procedure call. It is important to note that in case of the generic object, the token was residing to a different universe and was not migrated during the interactions.

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
AQT [ms]	107	110	104	114	112
AQHIT	1	1	1	1	1
AQMISS	0	0	0	0	0
RLT [ms]	39	38	38	42	40
CT [ms]	28138	68558	117506	219328	318981
TRAQ	0.0079	0.00753	0.007532	0.007524	0.007523

Table 15: Generic objects measurements for one client and acquire operation

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
AQT [ms]	0.061	0.051	0.0306	0.0714	0.0408
AQHIT	1	1	1	1	1
AQMISS	0	0	0	0	0
RLT [ms]	0	0	0	0	0
CT [ms]	15332	54575	106645	206131	306032
TRAQ	0.0333	0.0367	0.0263	0.0258	0.0279

Table 16: Private objects measurements for one client and acquire operation

In the same case of 30 node deployment, but where clients are located within the same universe as the private object, the measurements depicted in Figure 81 and Figure 82 were obtained. In case of the generic objects, the token resides in a different universe, thus the situation is the worst case scenario. In case of the private objects, the measurements show an increase in acquire time as the delay d decreases which denotes a higher number of acquire requests per time unit. The behavior is similar to the generic grid objects. As seen in Figure 82, there are a few cases where the acquire time is close to 0. These situations are those where the client node and the shared object are located on the same node (see previous measurements).

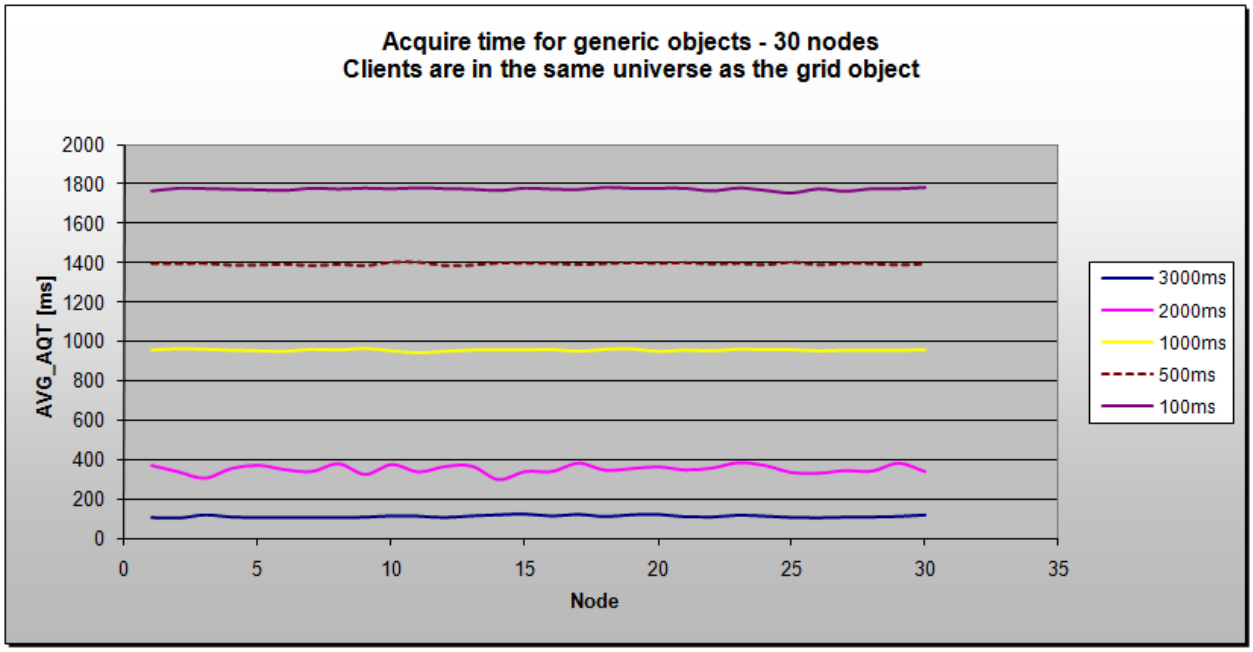

Figure 81: Acquire time for generic objects within the same universe

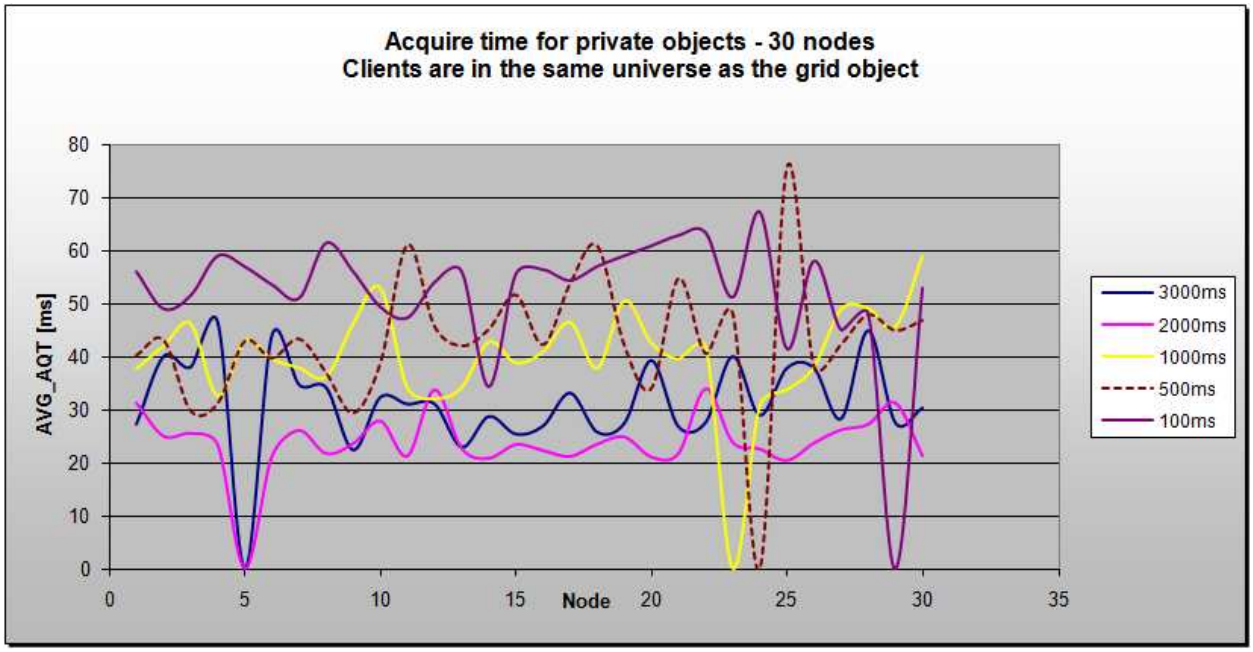

Figure 82: Acquire time for private objects within the same universe

. Figure 83 and Figure 84 illustrate the results where the clients are distributed across all universes for the same 30 node deployment per universe. In case of the generic objects, the same results were obtained as the ones presented in Section 7.2.2. Private objects show a uniform distribution across the nodes from all universes. The very low values correspond to the clients located on the nodes where the private object resides.

As can be seen from both graphs, the performance is comparable when the universe containing the generic object holds the token. If the token resides on a remote universe, the difference between private and generic objects becomes obvious when the delay value is set below 2000ms

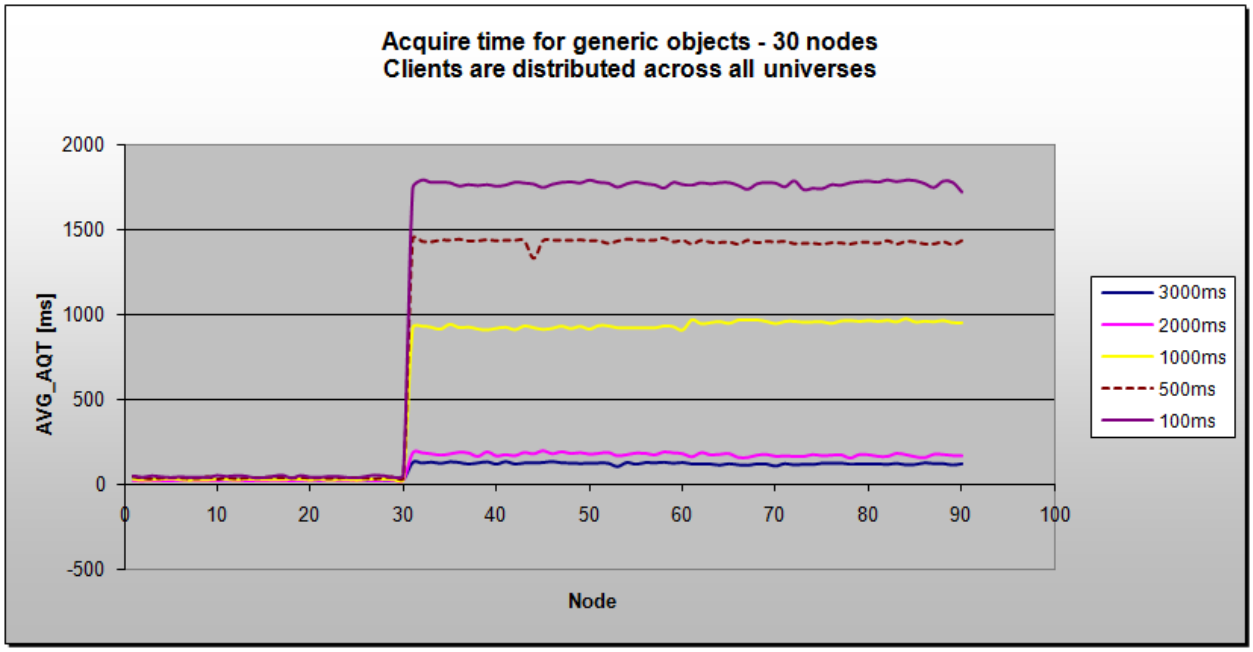

Figure 83: Acquire time for grid objects across all universes

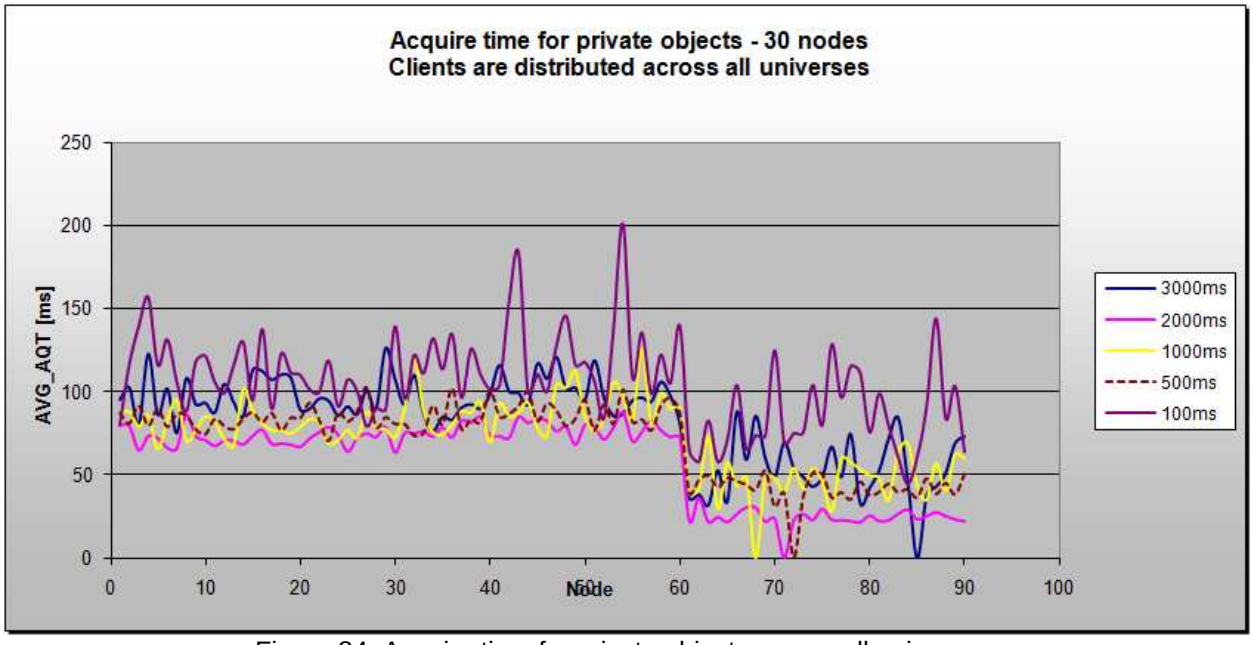

Figure 84: Acquire time for private objects across all universes

Discussion. One of the obvious differences in acquire time between private and generic objects is when the token does not belong to the same universe as the client node. The private object relies on the pure Java implementation of a multi-threaded monitor object. On the other side, the GUN implementation relies on a message queue which adds a queuing effect to all calls. Particularly to this

scenario is the fact that only acquire calls are being issued and no acquire exclusive calls are being made. As a result, the implementation of the private objects is more efficient. The saturation effect experienced when the request frequencies increase is expected to appear to the private objects too, but on a more moderate scale.

The second part of the experiment refers to ***acquire exclusive operations***. In case of a 30 node deployment where only one client is issuing requests and the client is located on the same node as the shared objects, the measurements presented in Table 17 and Table 18 were obtained. Similar to the other scenario, private objects exhibit lower acquire time values as well as shorter completion times. It is important to note that during the experiment, in case of the generic objects, the token migrates to the universe where it is requested from. As a result, the average value of AQET time decreases since all the following calls are within the universe containing the token. This explains why the average values for AQET are smaller than in case of AQT.

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
TT [ms]	118	110	120	116	121
AQT [ms]	34	35	33	35	37
AQHIT	1	1	1	1	1
AQMISS	0	0	0	0	0
CT [ms]	20266	60480	110283	210475	310787
TRAQ	0.018129	0.017036	0.017895	0.017467	0.016113

Table 17: Generic object measurements for one worker on acquire exclusive operation

Parameter	d=100ms	d=500ms	d=1000ms	d=2000ms	d=3000ms
TT [ms]	0	0	0	0	0
AQT [ms]	0.0204	0.0000	0.0612	0.0102	0.0612
AQHIT	1	1	1	1	1
AQMISS	0	0	0	0	0
CT [ms]	14712	54796	107711	204625	307267
TRAQ	0.0340	0.0337	0.0196	0.0363	0.0201

Table 18: Private object measurements for one worker on acquire exclusive operation

Figure 85 and Figure 86 show measurements for acquire exclusive time for generic and private objects where clients are located only within the same universe as the shared object. As in previous experiments, several variations of the delay d were chosen. An important remark is that the token migrates automatically to the universe where the clients are deployed after the first acquire exclusive call.

Comparing the two object types, one can notice similar patterns in acquire exclusive time depending on the delay d. Occasional fluctuations can be accounted on variable load of the cores which execute the node's logic. Besides the similar pattern, there are comparable values of acquire exclusive time for the same value of the delay d. It appears that the growth is initially slower for private objects, but the situation might reverse when the delay is set to lower values.

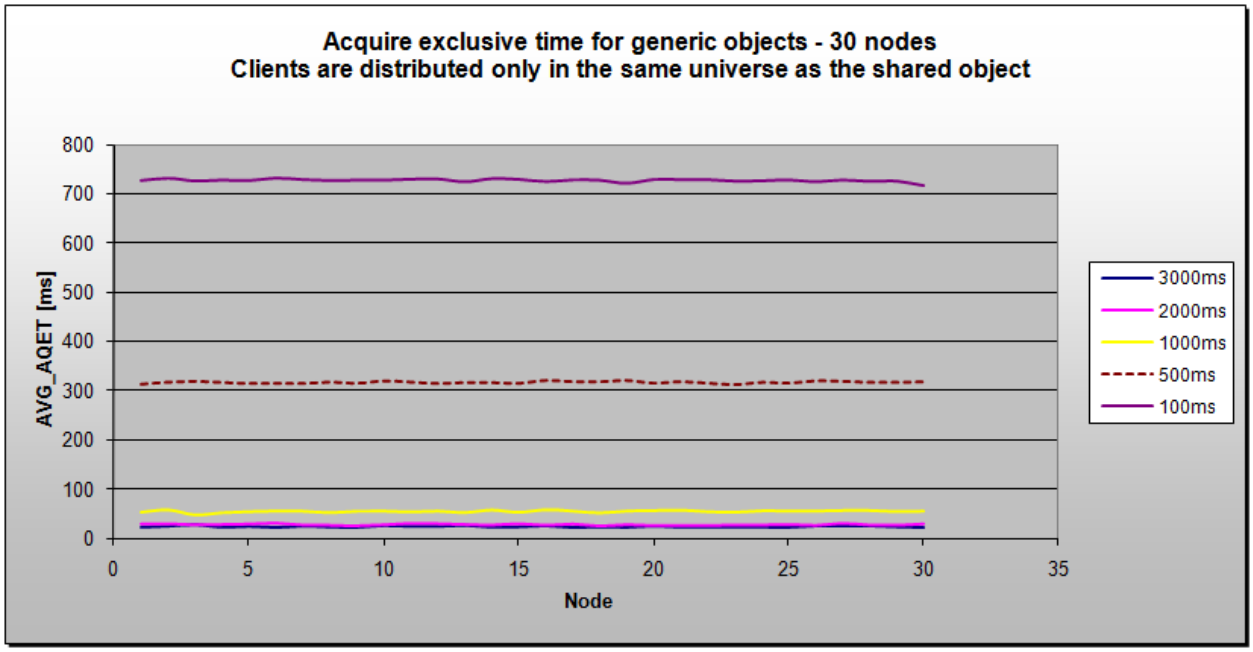

Figure 85: Acquire exclusive time for generic objects within the same universe

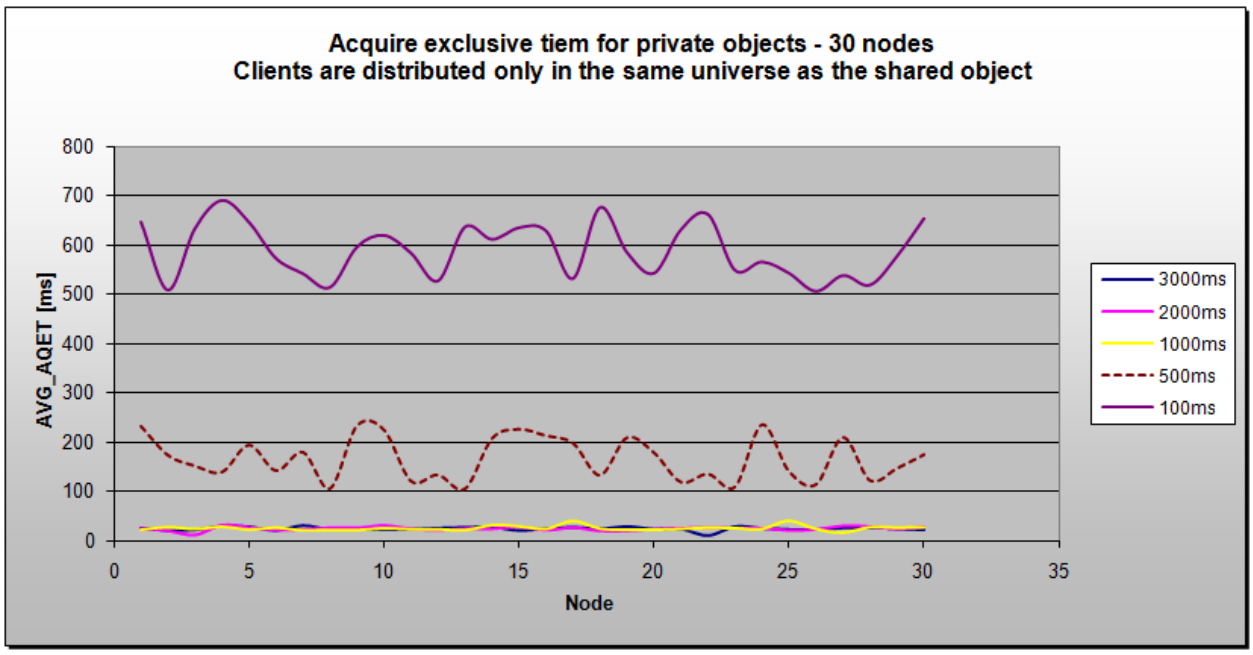

Figure 86: Acquire exclusive time for private objects within the same universe

Figure 87 and Figure 88 illustrate the measurements for acquire exclusive time for generic and private objects where clients are deployed across all universes. The results and behavior pattern obtained for generic objects is similar to the results presented in 7.2.3. Again, the global patterns of the two object types are similar.

Considering the more relaxed scenario where *d*=5000ms, the generic objects require about 600ms to satisfy an acquire exclusive request. Under the same conditions, the private objects require oscillating values around 1000ms. As the delay d decreases (more requests per time unit), generic

objects require about 1800ms in the worst case scenario where *d*=100ms. This time private objects require about 3500ms to satisfy a request.

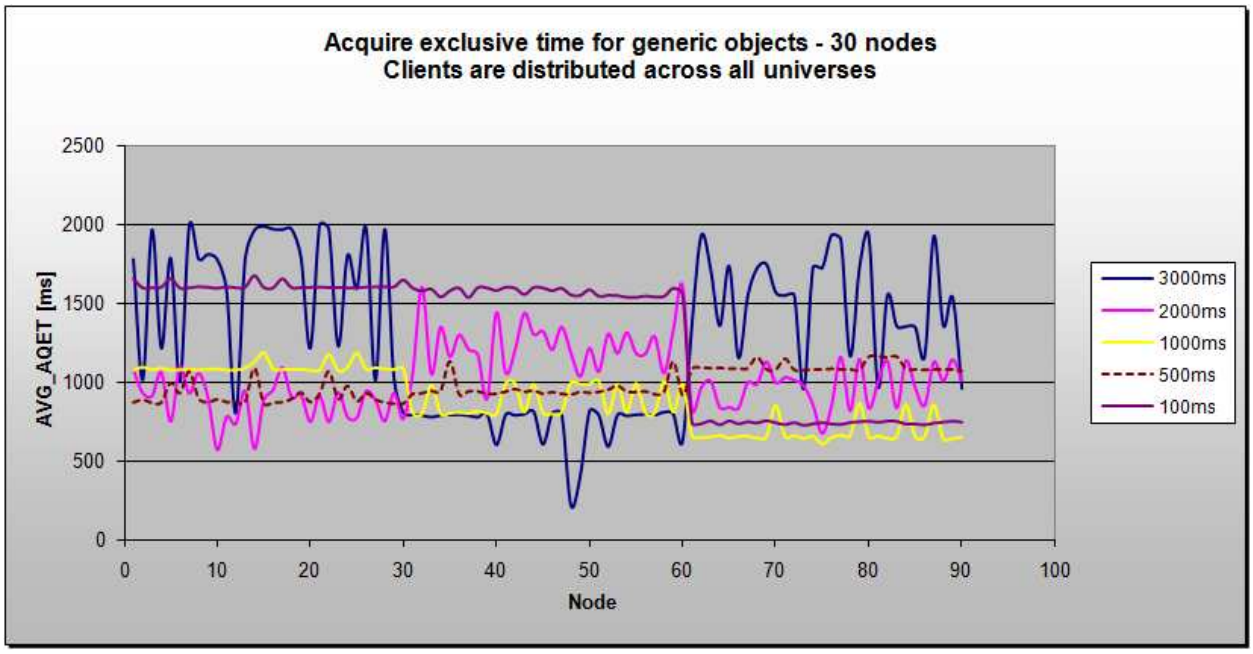

Figure 87: Acquire exclusive time for generic objects across all universes

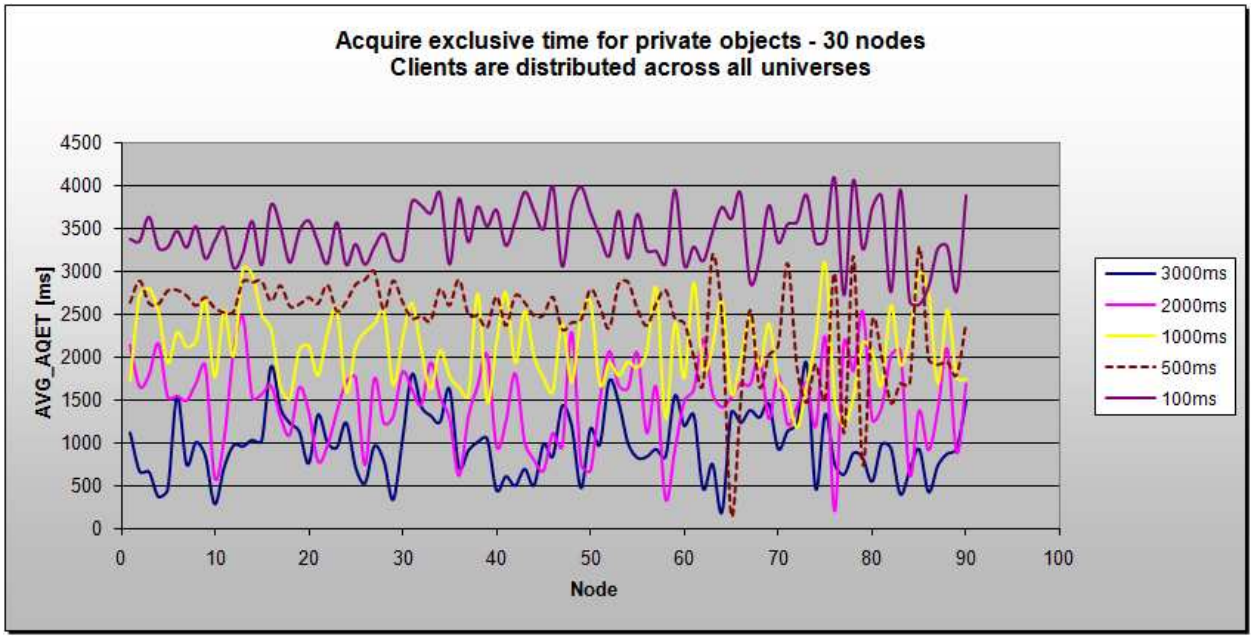

Figure 88: Acquire exclusive time for private objects across all universes

Discussion. In case acquire exclusive requests are issued, the bottleneck of the global sequencer (e.g. private object) becomes evident. The difference becomes evident when more than one client is used and accelerate when the number of client nodes increases. It is obvious that the GUN implementation is better in terms of response time and scalability than a plain sequencer. Although pure acquire operation appears to be more efficient, by extrapolating the results shown in this section, it is expected that seldom

acquire exclusive requests can cause severe performance degradation. As a conclusion, as expected from the model design, private objects are very efficient when the callers are localized in the proximity of the shared data and only a limited number of calls are being issued. Breaking the locality constraints as well as the request limit leads to performance degradation and it would be better to use generic objects for these situations.

7.2.6 Grid Migratory Objects

The experiments for migratory objects which are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes. As observed during the experiment execution, the most relevant aspects were revealed in the maximum node configuration.

Figure 89 shows acquire exclusive time for migratory objects for different values of the remote method execution cost. Basically there is no dependency to this parameter since the method execution cost does not interfere with granting access to the shared object. The same applies for the release operation where an asynchronous solution was adopted too. In case of different node deployment configurations (e.g. 5, 10, 20, 30), the same acquire exclusive time values were obtained independent on the number of nodes. This shows that the migratory object implementation is scalable in respect to the grid universe.

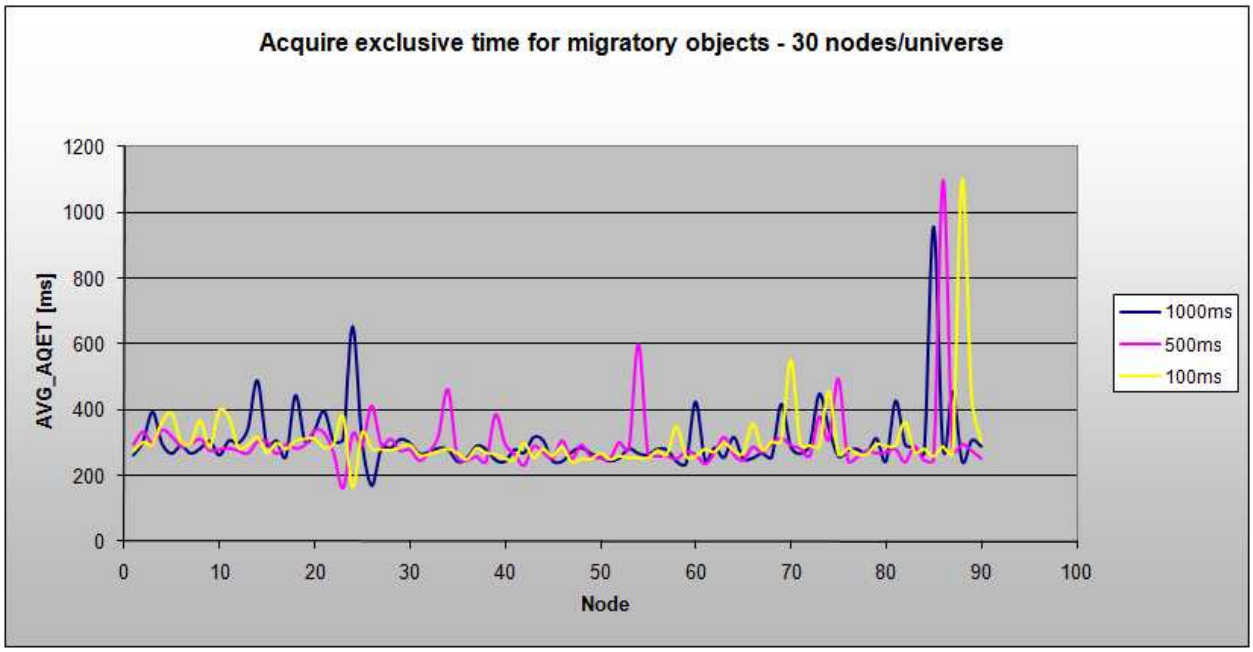

Figure 89: Acquire exclusive time for migratory objects

Figure 90 and Figure 91 show the differences between migratory and generic grid objects in respect of the acquire exclusive time for different values of the parameter *d*. As seen in both situations, there is a difference of about 100ms in favor of the generic objects. One must acknowledge that this experiment is a corner case which aims to answer the question whether there are situations where migratory objects are a real advantage compared to generic objects (assuming the same interaction orchestration is defined). It is self-evident that migratory objects would perform better than generic objects

when the latter ones are not replicated. In this case, generic objects would experience the bottleneck effect.

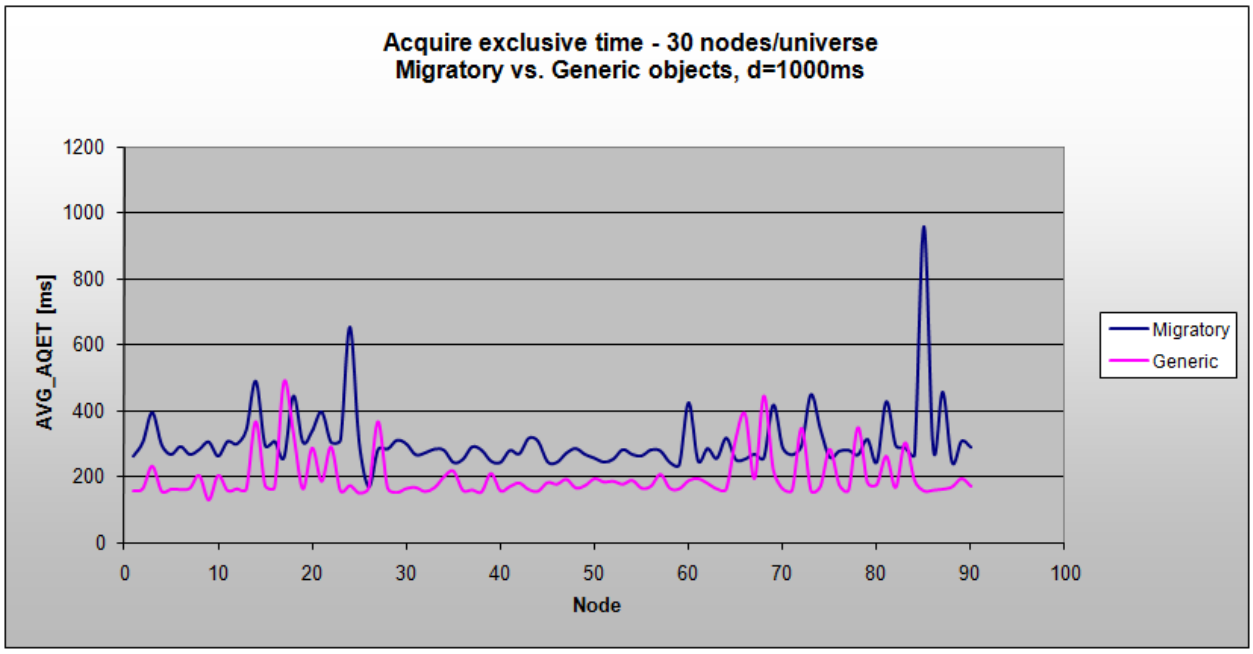

Figure 90: Acquire exclusive time: migratory vs. generic objects, d=1000ms

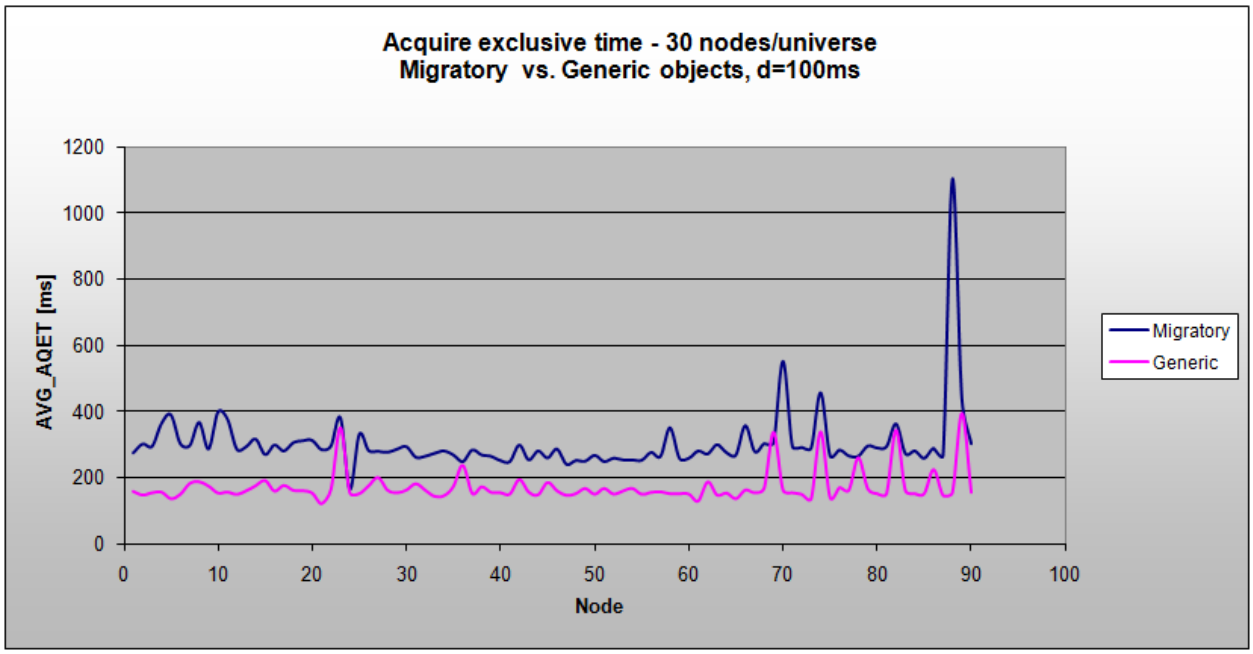

Figure 91: Acquire exclusive time: migratory vs. generic objects, d=100ms

The opposite case is the one described in the experiment, where generic objects are highly replicated. The experiment has proved that migratory objects do not improve the performance by the locality aspect since the locality is already offered by the generic object without the migration penalty. However the difference is not significant. Most probably an even point would be the situation where generic objects are replicated just on a certain number of nodes. Exceeding that threshold would bring

the system in the state described above. Below that threshold it is expected that migratory objects perform better than the generic ones.

Figure 92 shows the differences between migratory and generic grid objects in respect of completion time. The differences seen in case of acquire exclusive time propagate to the application completion time. As pointed out above, release time does not play a role to the global completion time.

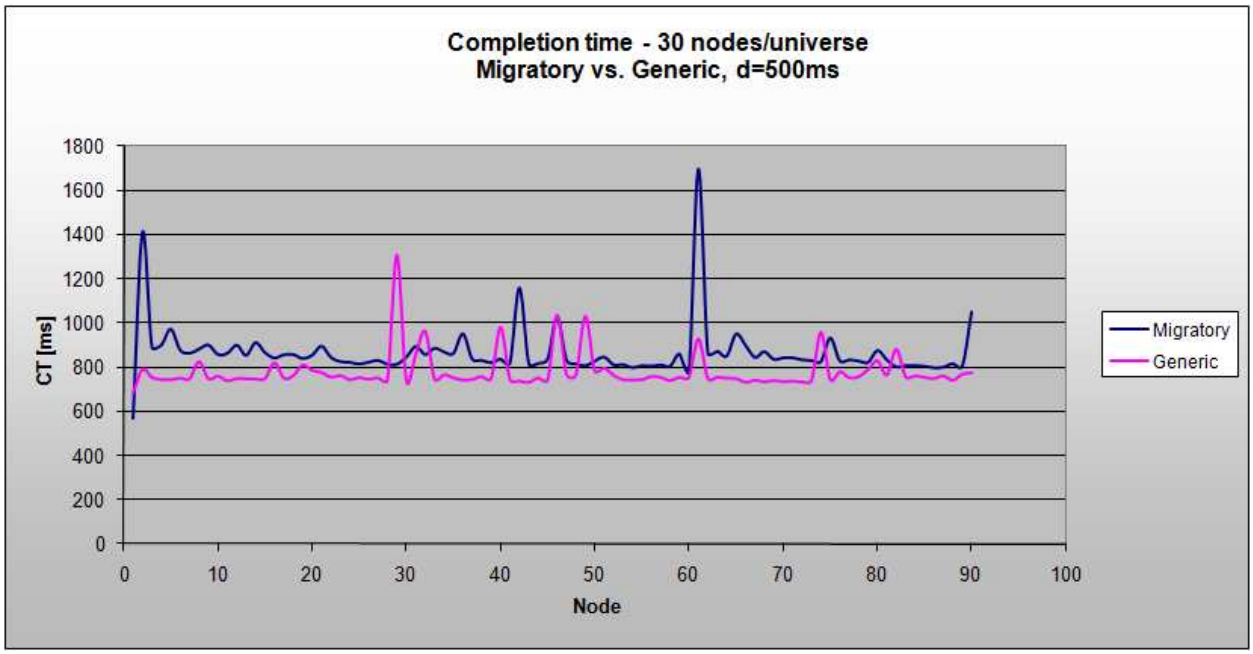

Figure 92: Completion time: migratory vs. generic objects

7.2.7 Grid Producer-Consumer Objects

The experiments for producer-consumer objects which are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Similar to the previous cases, due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

One important aspect for producer-consumer objects is related to the object weight. In the grid shared object model, objects have weights associated which denote the object size (e.g. memory). The object weight implies the cost to transport an object from one node to another. The higher the weight is the higher the object transfer time. To simplify things object size units were associated to the object weight. During the experiments, different object size units were used and it was considered that one object size unit corresponds to a transport penalty of 10ms.

In the simplest situation where there is only one producer and one consumer, and the delay between subsequent operations was *d*=5000ms, and the “one object per node” replication policy was chosen, the results summarized in Table 19 were obtained. In this case, for a 10 object units (ui), one can notice that the acquire exclusive time has comparable values for producer-consumer and generic objects. The release time for producer-consumer objects is significantly higher due to the write-update protocol. The acquire time for producer-consumer is almost three times smaller than for the generic objects

because the object synchronization cost does not appear anymore. Summing these up, the completion time for producer-consumer objects is smaller than in case of the generic objects.

Parameter [ms]	**P-C object**	**Generic object**
OST	175	173
AQET	991	1044
RLT	208	28
AQT	111	272
RT	42	34

Table 19: Producer consumer vs. generic objects for one consumer

Figure 93 and Figure 94 illustrate the differences between producer-consumer and generic objects in respect of acquire and completion time, if one node per universe replication is used and the object size unit is equal to 1, for a delay of 10000ms. The overall graph shape is similar to the ones obtained in Section 7.2.2 where the first group of nodes belongs to the universe holding the token. As shown in both graphs, producer-consumer objects exhibit lower acquire and completion time values than the generic objects

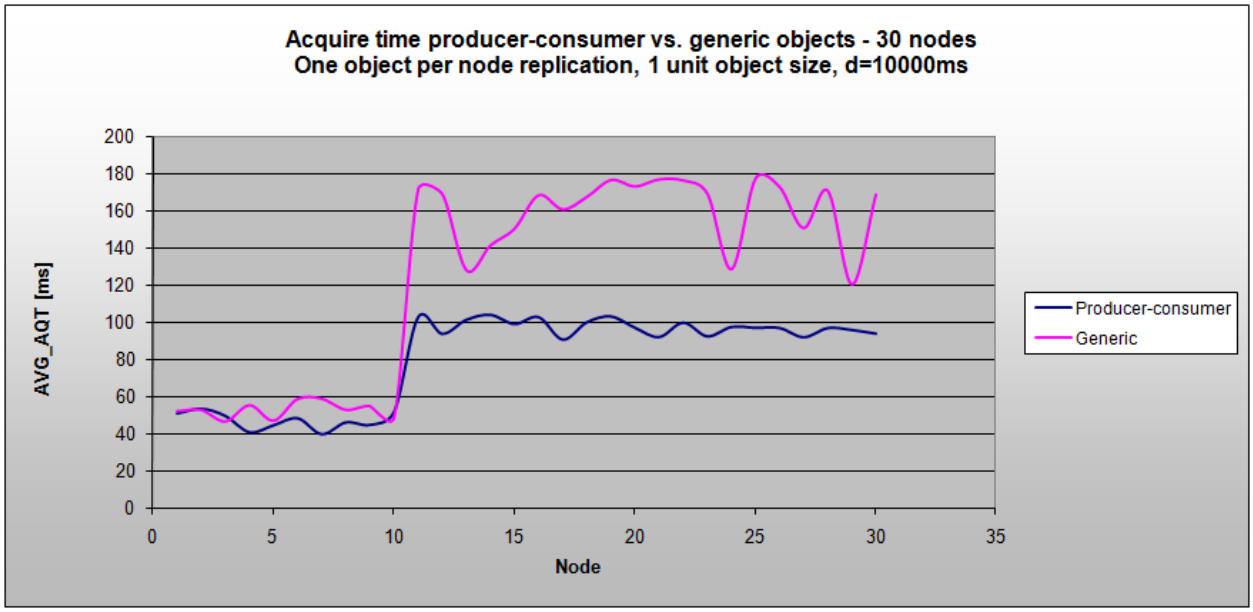

Figure 93: Acquire time: producer consumer vs. generic objects

Table 20 summarizes the differences between producer-consumer and the generic object in respect of the OST, AQET and RLT parameters. As expected, AQET is smaller in case of the producer-consumer since the chances for object synchronization prior to granting the access are totally reduced. The opposite situation can be noticed for the release time (RLT) because the release operation is synchronizing all object replicas. It is important to note that for the entire application execution, the distribution in time of these operations is important since a snapshot to the system's status does not reflect the entire behavior of the system.

Parameter [ms]	Object type	AVG time value [ms]	Count
OST	pc_object	59	2980
AQET	pc_object	281	100
RLT	pc_object	1783	100
OST	generic_object	22	2982
AQET	generic_object	1051	100
RLT	generic_object	25	100

Table 20: Producer consumer vs. generic objects for 30 clients

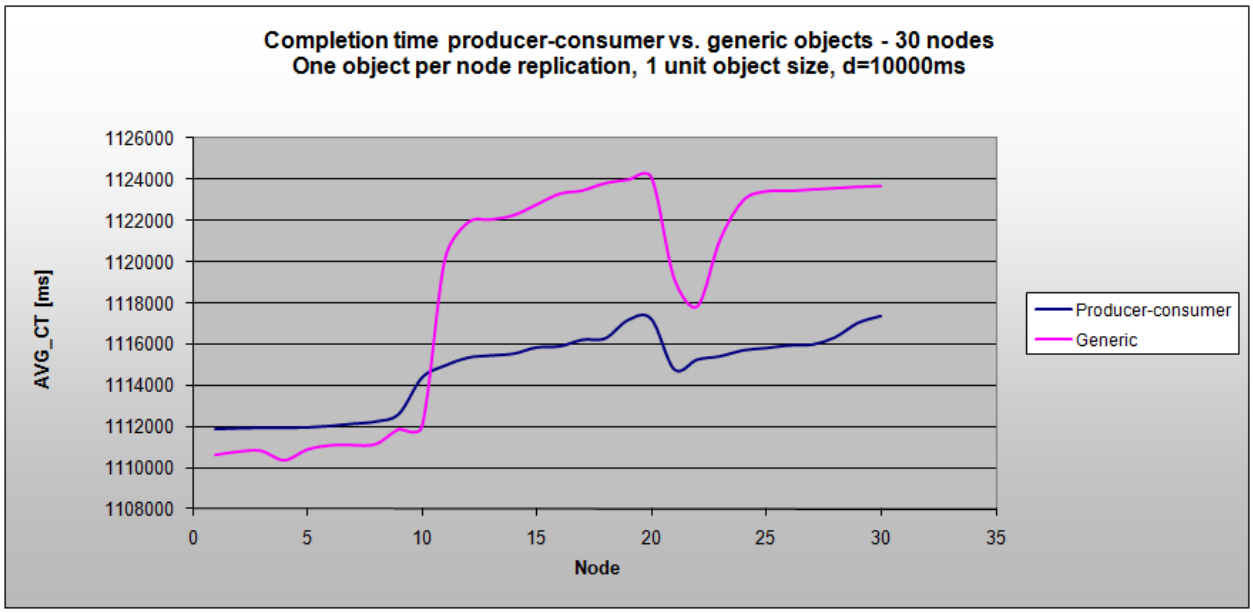

Figure 94: Completion time: producer consumer vs. generic objects

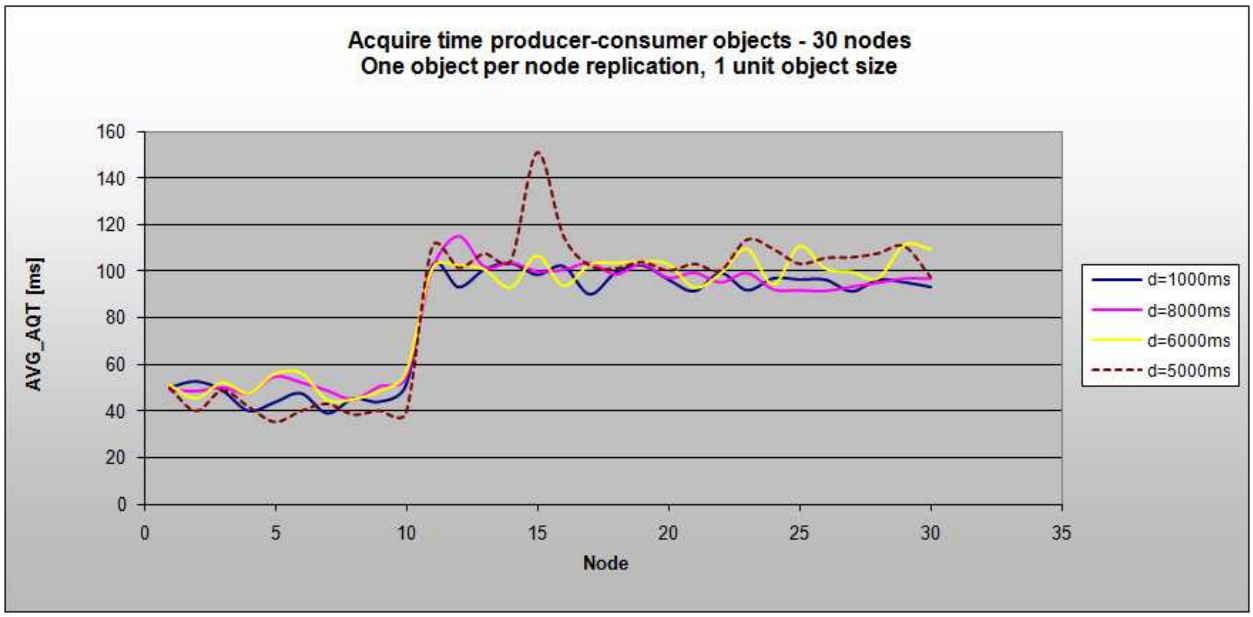

Figure 95: Acquire time: producer consumer vs. generic objects for different timings

As specified in the experiment's description, different delays between the acquire requests have been selected. Figure 95 shows that independent on the selected acquire request frequencies, the same values and pattern of the acquire time is obtained. As in the previous situations, the first group of nodes belongs to the universe holding the token and experiences a lower acquire time. Table 21 shows the differences in respect of the OST, AQET and RLT parameters depending on the parameter d. As expected, OST is the same in all cases since it is the same kind of object which is being synchronized. As the number of requests increases per time unit (d is decreasing), a slight increase in terms of the AQET parameter is noticed. This can be explained by the increasing saturation of the primary node's event queue. Release time appears to be constant independent on the selected delay. In case of a stable system (one producer and no queue overflow) the node holding the modified object has to update every time the same set of consumer nodes. As the consumer node set is fixed, the value for the RLT time is constant.

Parameter [ms]	**d=10000ms**	**d=8000ms**	**d=6000ms**	**d=5000ms**
OST	59	58	58	59
AQET	281	335	353	318
RLT	1783	1781	1781	1793

Table 21: Producer consumer vs. generic objects for 30 clients and different timing

In case the object size increases, it was only considered a ten times increase, from 1 ui to 10 ui. Figure 96 shows that an increase in object size leads to an increase in acquire time. Table 22 summarizes OST, AQET and RLT. It can be noticed that an increase in object size leads to a higher average object synchronization time as well as to higher AQET and more dramatically RLT parameter values. The explanation for this behavior is that increasing the object size increases the release time and thus delaying the moment of granting the acquire request. If different delays are used in the producer and consumer (experimentally it was considered delta = d/2) the above behavior does not appear anymore and the acquire time (both exclusive and non-exclusive) do not depend on size of the shared object.

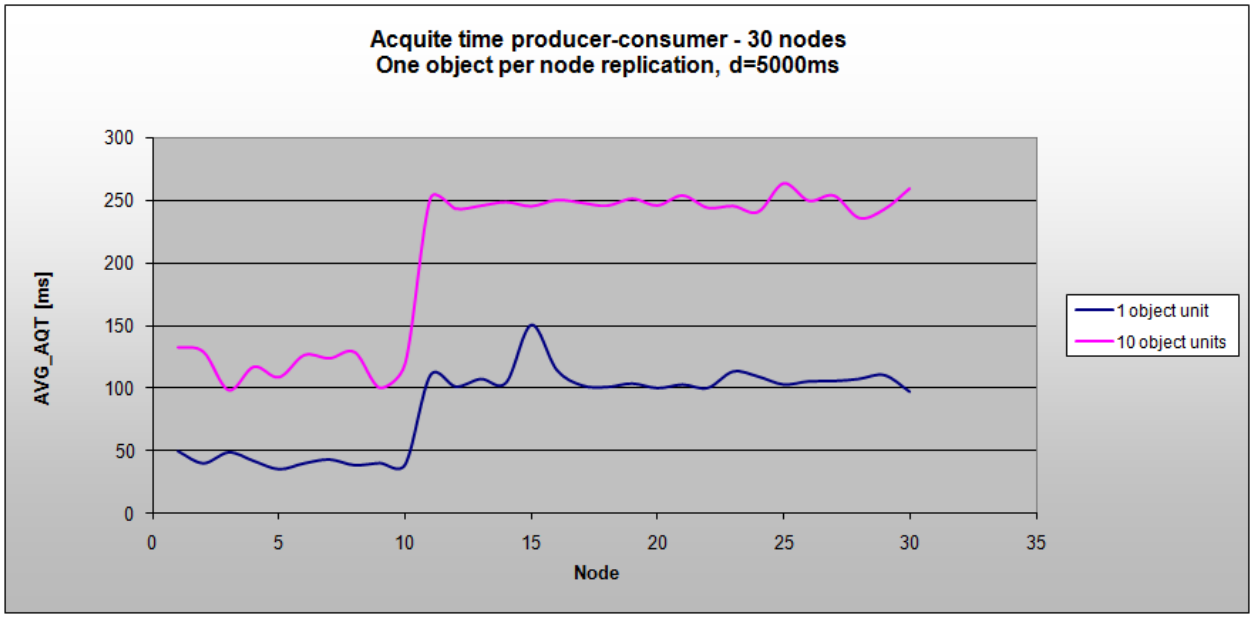

Figure 96: Acquire time: producer consumer vs. generic objects for different object size

Parameter [ms]	1 object unit	10 object unit
OST	59	147
AQET	318	803
RLT	1793	4341

Table 22: Producer consumer vs. generic objects for 30 clients and different object size

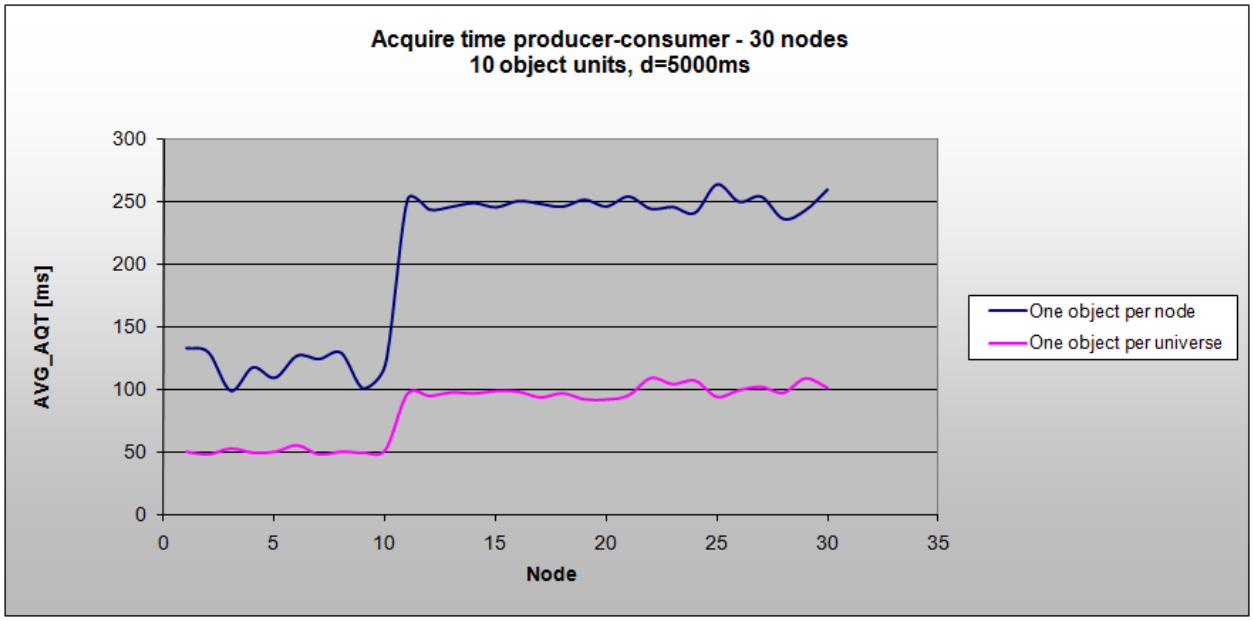

Figure 97: Acquire time: producer consumer vs. generic objects with different replication

In case “one object per universe” replication is used instead of the standard “one object per node”, AQT values experience a significant reduction as shown in Figure 97. In both cases similar OST and AQET parameter values were measured, as presented in Table 23. Since there is a significant reduction in the number of replicated objects, the RLT parameter is dramatically reduced when “one object per universe” replication rule is used. As a consequence, the overall application completion time reduces too.

Parameter [ms]	One object/node	One object/universe
OST	147	169
AQET	803	754
RLT	4341	362
CT	629291	610972

Table 23: Producer consumer vs. generic objects for 30 clients and different replication

In the discussions of the previous paragraphs a number of 30 clients were considered. The same experiments were run in a configuration of 30, 21, 15, 9 and 1 client. Acquire time dependency to the number of clients is shown in Figure 98. One can notice that virtually the same values are obtained for acquire time independent on the number of clients.

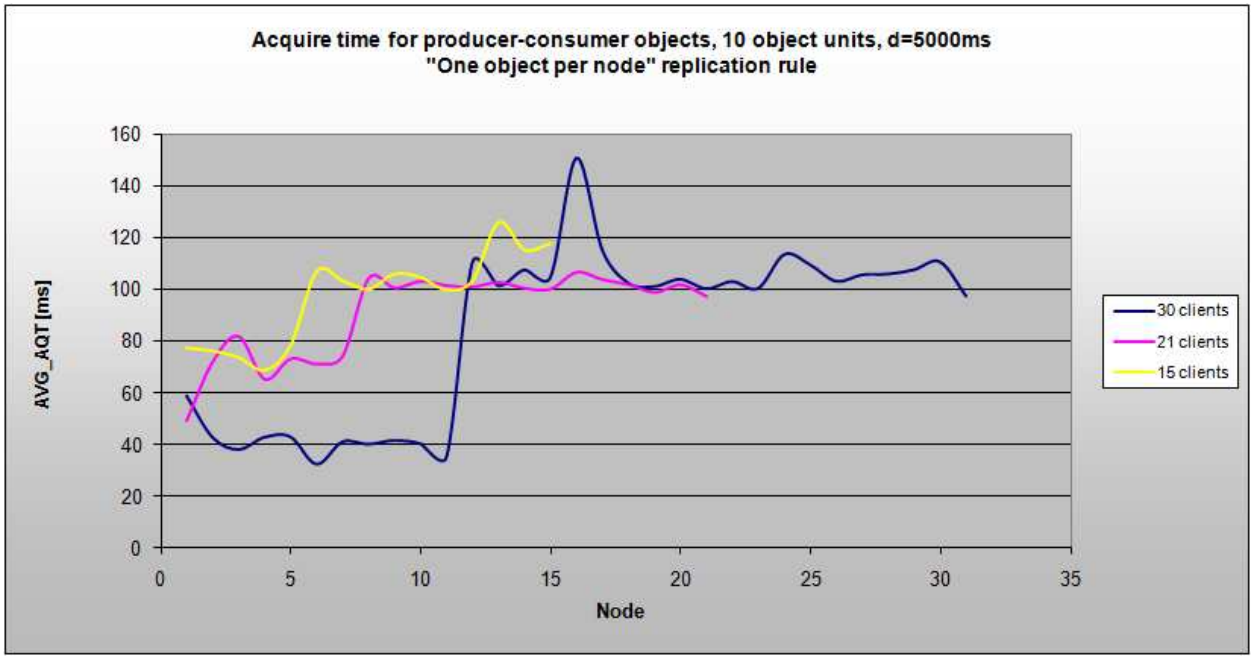

Figure 98: Acquire time: producer-consumer vs. generic objects with different client counts

In case *d*=5000ms, and one object size unit and "one object per node" replication rule, Table 24 shows the dependency of OST, AQET and RTL in respect of the client count. As expected, OST is constant since the same kind of object is being synchronized. It can be noticed that AQET slightly decreases together with the number of clients (acquire queue effect) and the release time increases together with the number of clients since a higher number of object replicas need to be updated.

Parameter [ms]	**30 clients**	**21 clients**	**15 clients**	**9 clients**	**1 client**
OST	110	167	165	147	154
AQET	1210	870	682	756	980
RLT	4340	1247	914	1235	110

Table 24: Producer-consumer vs. generic objects for 30 clients and different replication

7.2.8 Grid Read-Mostly Objects

The experiments for read-mostly objects which are described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Similar to the previous cases, due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

Considering a stable system where requests are being issued every d=10000ms where there are 10 consumers per universe and only one producer in the entire grid universe and one object per node replication rule is chosen, as shown in Figure 99, read-mostly objects show a much lower acquire time. As summarized in Table 25, acquire exclusive time as well as release times are higher due to the write-update protocol which is used for read-mostly objects. The graphic shape is similar to the acquire time for generic objects where the first group of nodes belong to the universe holding the token. Since there is only one producer there is no token movement and *TT*=0.

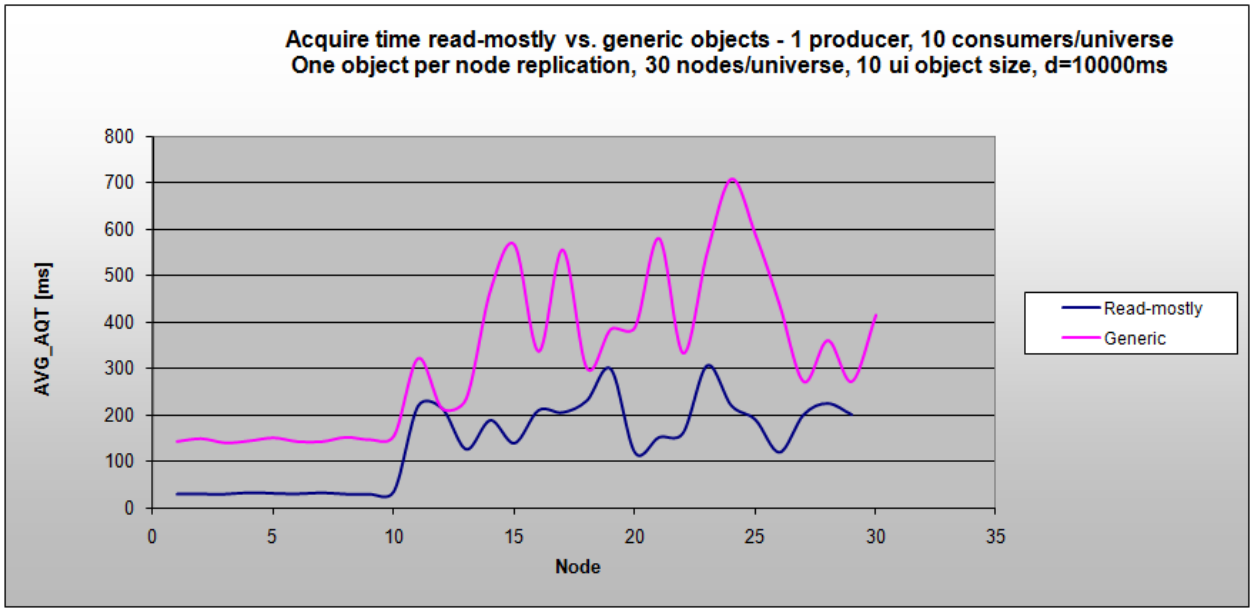

Figure 99: Acquire time: read-mostly vs. generic objects

Parameter	OST [ms]	AQET [ms]	RLT [ms]	CT [ms]
read-mostly	149	1219	4513	4500262
generic	115	850	41	4500287

Table 25: Read-mostly vs. generic objects for 30 clients

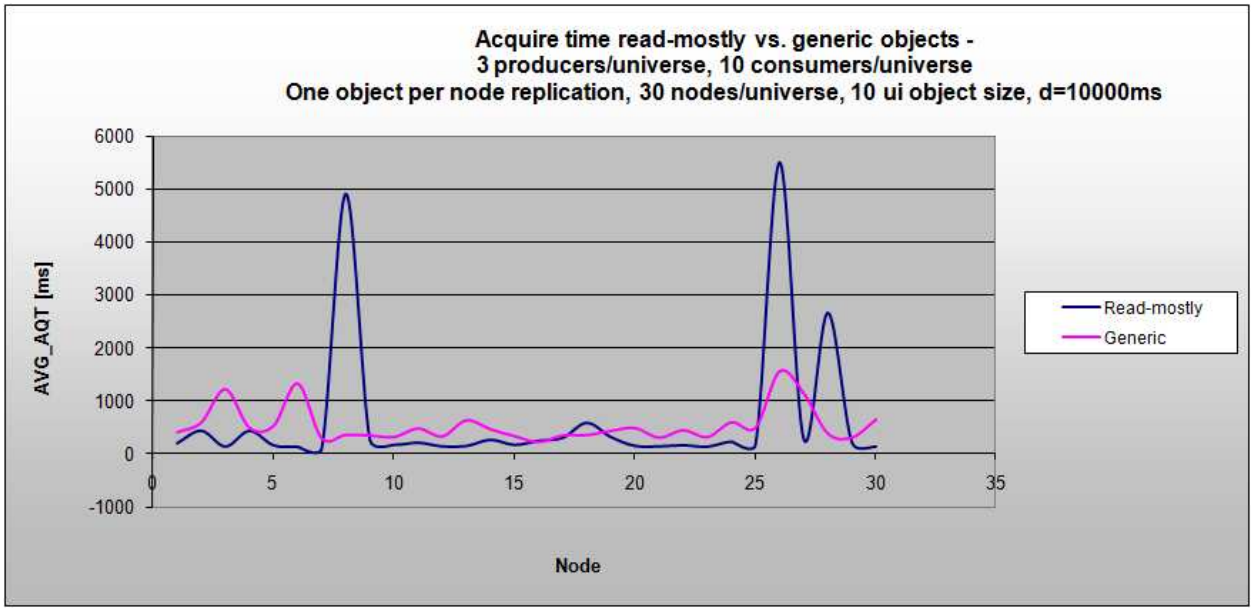

Figure 100: Acquire time: read-mostly vs. generic objects for 3 producers and 10 consumers

Parameter	OST [ms]	TT [ms]	AQET [ms]	RLT [ms]	CT [ms]
read-mostly	148	1110	1178	5646	6000292
generic	139	1112	1798	40	6000276

Table 26: Read-mostly vs. generic objects for 3 producers and 10 consumers

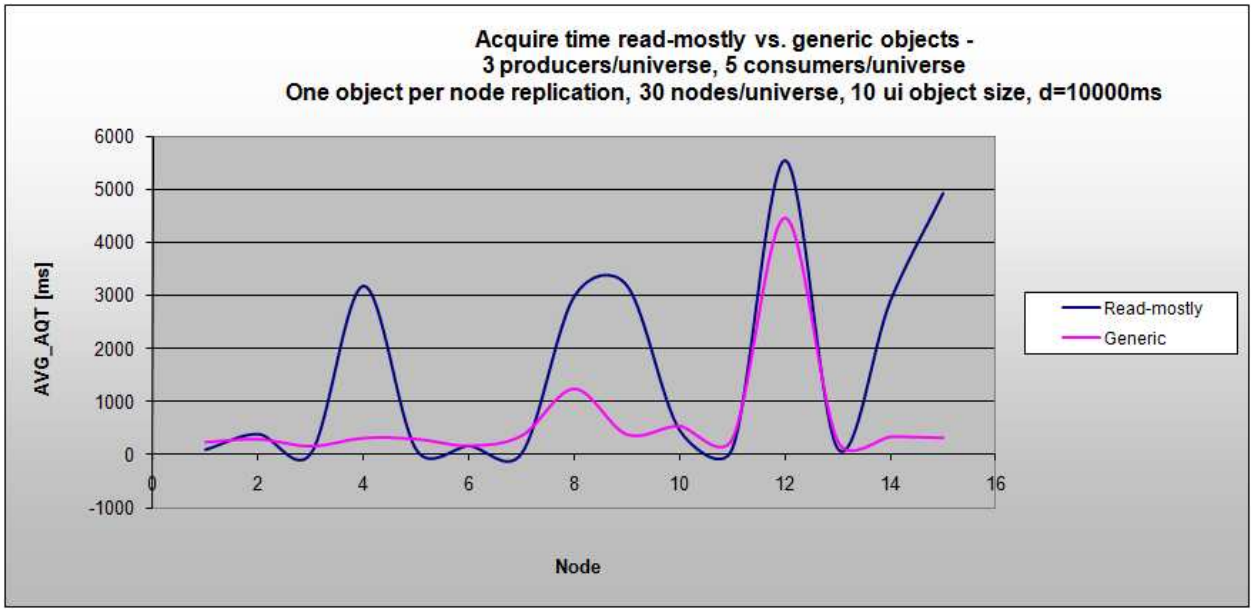

Figure 101: Acquire time: read-mostly vs. generic objects for 3 producers and 5 consumers

Parameter	OST [ms]	TT [ms]	AQET [ms]	RLT [ms]	CT [ms]
read-mostly	148	1088	1207	3461	3500237
generic	139	1049	1494	40	2450300

Table 27: Read-mostly vs. generic objects for 3 producers and 5 consumers

For the same configuration of 30 nodes per universe, “one object per node” replication rule and *d*=10000ms, the variation of acquire time is shown in Figure 100 and Figure 101 where the producer-consumer ratio is 3:10, respectively 3:5. While the former situation depicts a stable system (occasional spikes are caused by delays caused by write updates), the later shows the starting point of an unstable system as the producer-consumer ratio is increasing and more nodes are producing data rather than consuming. The other significant parameters are summarized in Table 26 and Table 27. Both situations show the same parameter variation pattern. It is important to note that the release time is higher in the first case since the number of client nodes which have to be synchronized is double (10 instead of 5).

For the same configuration of 30 nodes per universe, “one object per node” replication rule and 3 producers/universe and 10 consumers/universe, by changing the frequency of the acquire operations, the results depicted in Figure 102 and Table 28 were obtained.

d	OST [ms]	TT [ms]	AQET [ms]	RLT [ms]	CT [ms]
10000 ms	148	1110	1178	5646	6000292
7000 ms	145	1101	1502	5588	4200288
5000 ms	146	1223	3451	5580	3000287

Table 28: Parameter variation for read-mostly objects depending on acquire frequency

It can be noticed that for d=10000ms and d=7000ms the system exhibits the same performance parameters. Increasing the rate of the acquire requests by setting d to 5000ms, there is a dramatic increase in acquire time, as well as in the acquire exclusive time as a result of the primary node’s increasing congestion. As expected, the release time remains constant since the same operations are

performed independent on the acquire frequency (the same number of objects are being updated and the same primary node is contacted to release the lock on the shared object).

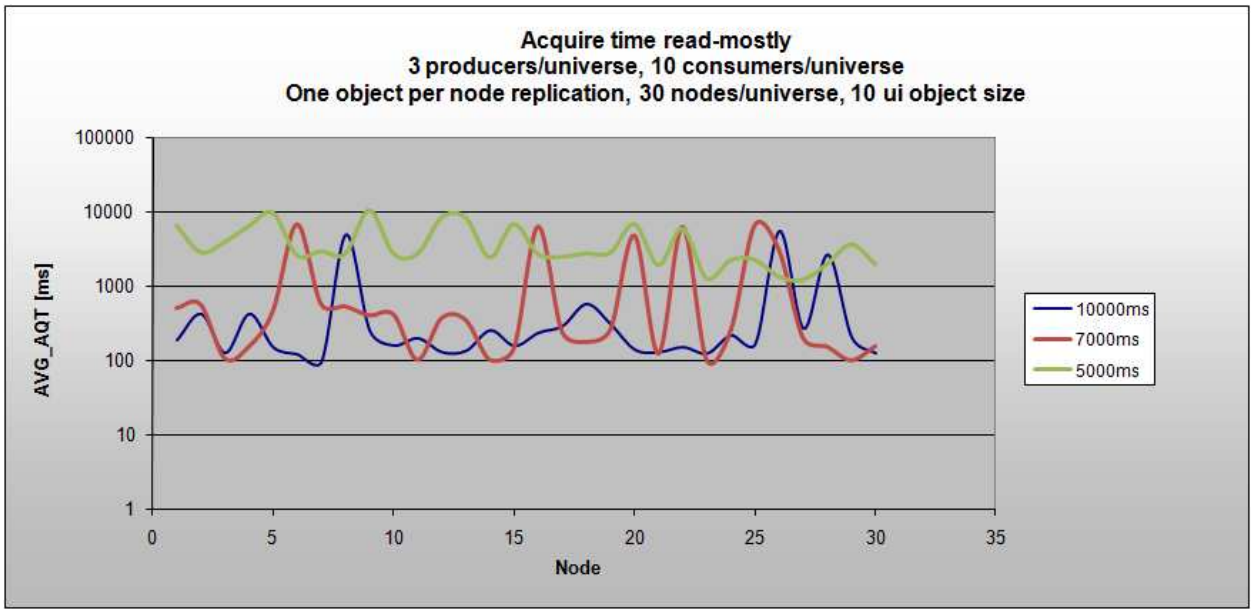

Figure 102: Acquire time variation for read-mostly objects depending on acquire frequency

Figure 103 and Table 29 show the parameter values for both “one object per node” and “one object per universe” replication rules, for the same configuration of 30 nodes per universe, *d*=10000ms and 3 producers/universe and 10 consumers/universe.

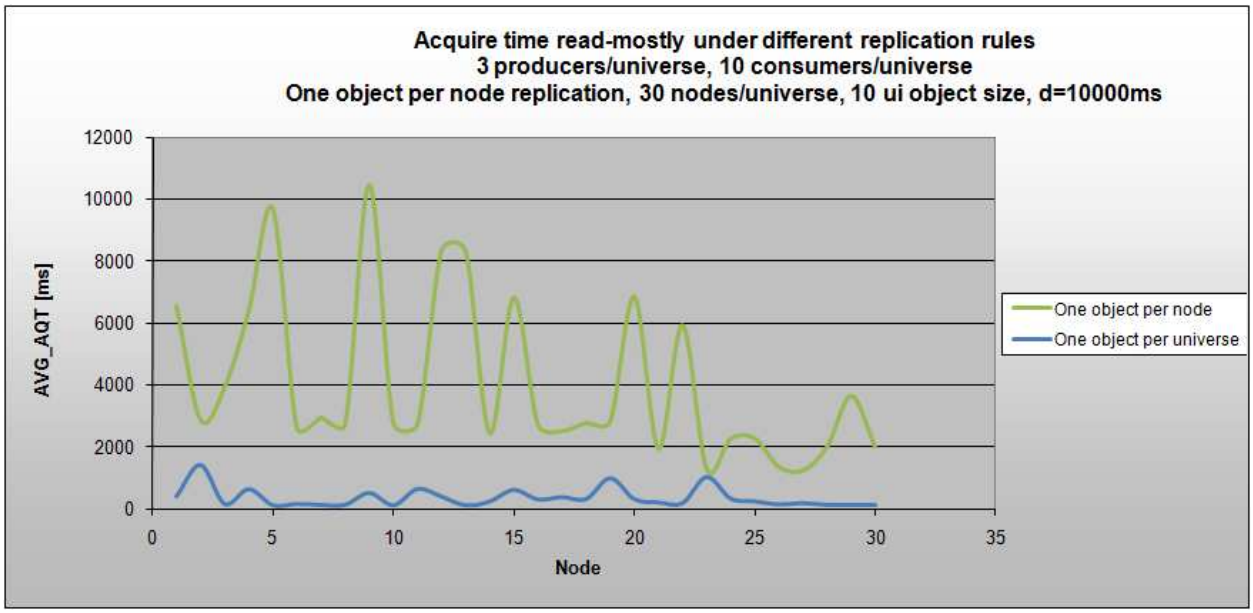

Figure 103: Acquire time variation for read-mostly objects depending on replication rules

Replication	OST [ms]	TT [ms]	AQET [ms]	RLT [ms]
One/node	146	1223	3451	5580
One/universe	170	1103	1234	380

Table 29: Parameter variation for read-mostly objects depending on replication rules

As the total number of replicated objects decreases, there is decrease in terms of acquire and acquire exclusive time, as well as the release time (since less objects are being synchronized upon releasing the shared object). In the experiments the object size has been chosen quite modest meaning that the object can be read remotely very fast. For these situations it appears that choosing “one object per universe” replication rule is more effective than having the object replicated on all the nodes. The overhead of replica management and object synchronization takes a significant amount of computing time, thus for small objects is it more effective to use as less replicas as possible. The situation changes as the object size increases especially if many remote operations are being issued. However, as this situation is not the main performance aspect of the experimental evaluation (this relates more to the proper replication selection), this aspect is not being investigated in more details in this section.

Another scenario of the experimental evaluation considered different producer/consumer ratios. A stable system was considered when only one producer was used in the entire grid universe and 10 consumers for every universe were deployed. The request frequency was determined by d=10000 ms. Figure 104 and Table 28 present the acquire time as well as the OST, TT, AQET and RLT parameters.

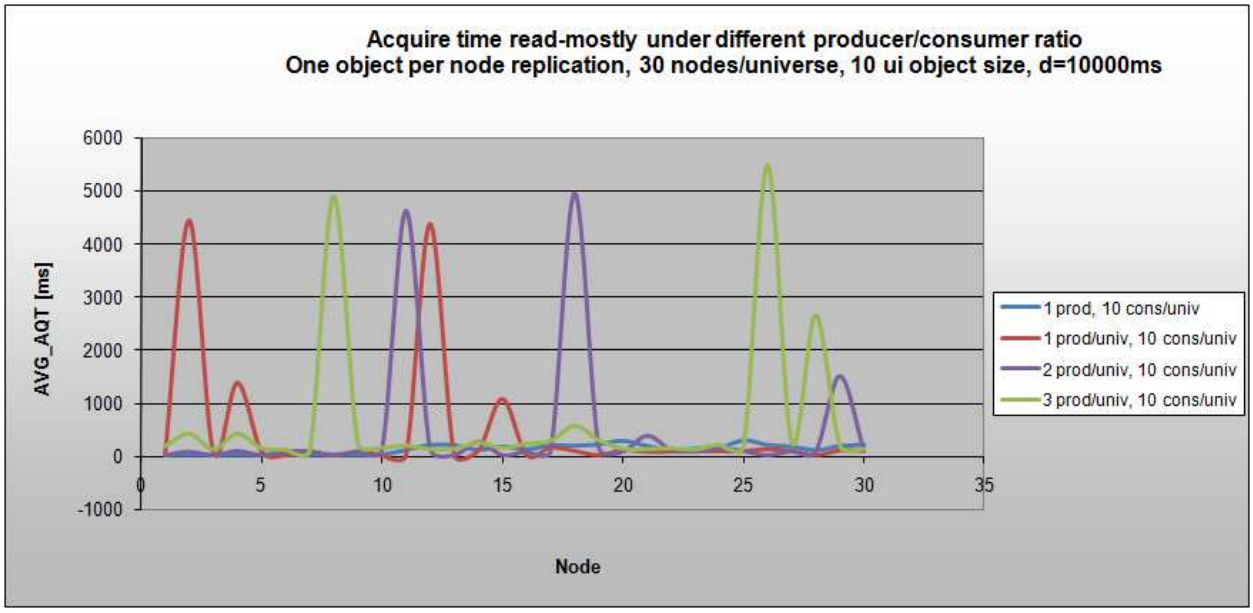

Figure 104: Acquire time variation for read-mostly objects depending on producer/consumer ratio

Variant	OST [ms]	TT [ms]	AQET [ms]	RLT [ms]
1 prod, 10 cons/univ	149	0	1219	4513
1 prod/univ, 10 cons/univ	131	757	787	4790
2 prod/univ, 10 cons/univ	149	753	2113	5191
3 prod/univ, 10 cons/univ	146	1223	3450	5580

Table 30: Parameter variation for read-mostly objects depending on producer/consumer ratio

Acquire time exhibits a steady variation with occasional spikes due to the synchronizations between producers and consumers. Experimentally, by choosing different acquire frequencies for producers and consumers, it was noticed that such spikes do not occur anymore. In terms of the acquire exclusive time, its value increases together with the number of producers since the token competition is increasing and thus inducing higher waiting times. Interestingly the release time remains constant since GUN implements an asynchronous release operation. Since the number of consumer nodes is constant

(in this experiment was set to 10), the release time is approximately the same since the same number of object replicas are being updated.

7.2.9 Grid Result Objects

The experiments for result objects described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Similar to the previous cases, due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per and a number of 3 universes.

Considering a relatively stable system where requests are being issued every d=5000ms where there are 10 producers and only one consumer, as represented in Figure 105, acquire exclusive time is much smaller for result objects. This is a natural consequence of the fact that there are virtually no locks in the acquire exclusive implementation. On the opposite side, the generic objects show the same behavior presented in previous sections. Table 31 shows that the acquire time is almost 6 times higher for the result objects as an effect of the object composition procedure out of the disjoint parts. As there is no token movement in this scenario, the token time (*TT*) is zero for result objects. The penalty of the higher object synchronization time is generated by the higher complexity of the object composition procedure and probably on the higher simultaneous updates which causes a longer execution time of the synchronization procedure.

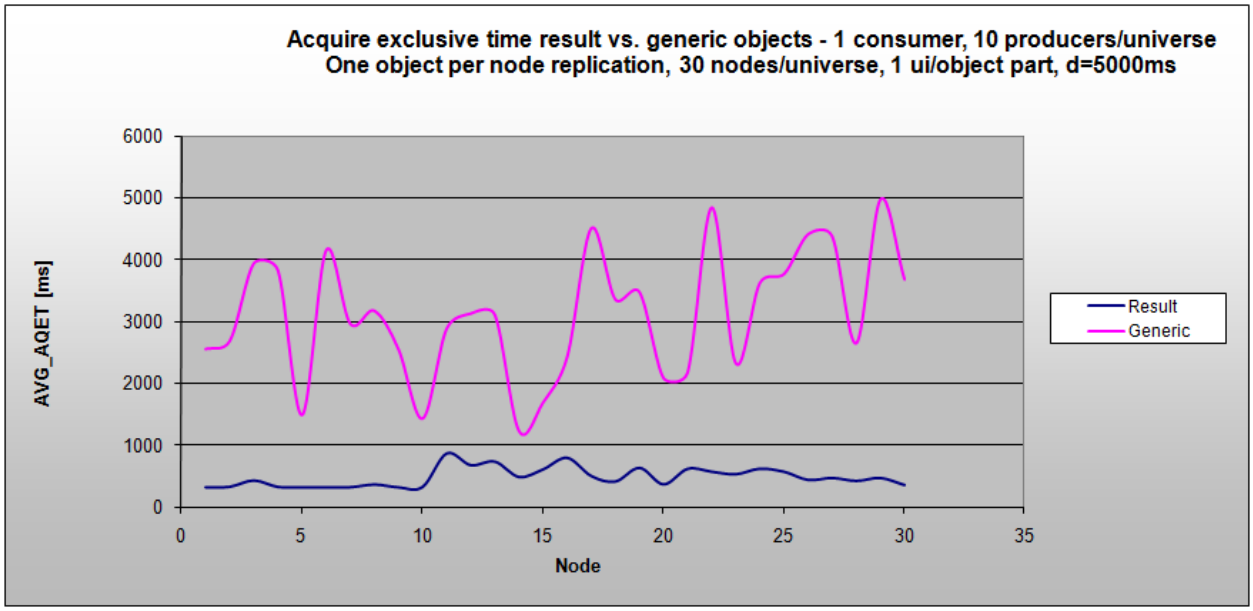

Figure 105: Acquire exclusive time: result vs. generic objects for 1 consumer

Parameter	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]	CT [ms]
result	1454	0	1965	37	350292
generic	438	782	355	48	350263

Table 31: Result vs. generic objects for 1 consumer

While increasing the number of consumers from one to three and keeping the same experiment parameters as in the previous case, it can be observed that the system made out of result objects is still in a relatively stable state where AQET is below 1000 ms (see Figure 106 and Table 32). While the

system made out of generic objects is moving towards the unstable state (increasing AQT, TT), the result objects scenario shows a still stable system (same AQET, AQT values as in the previous case).

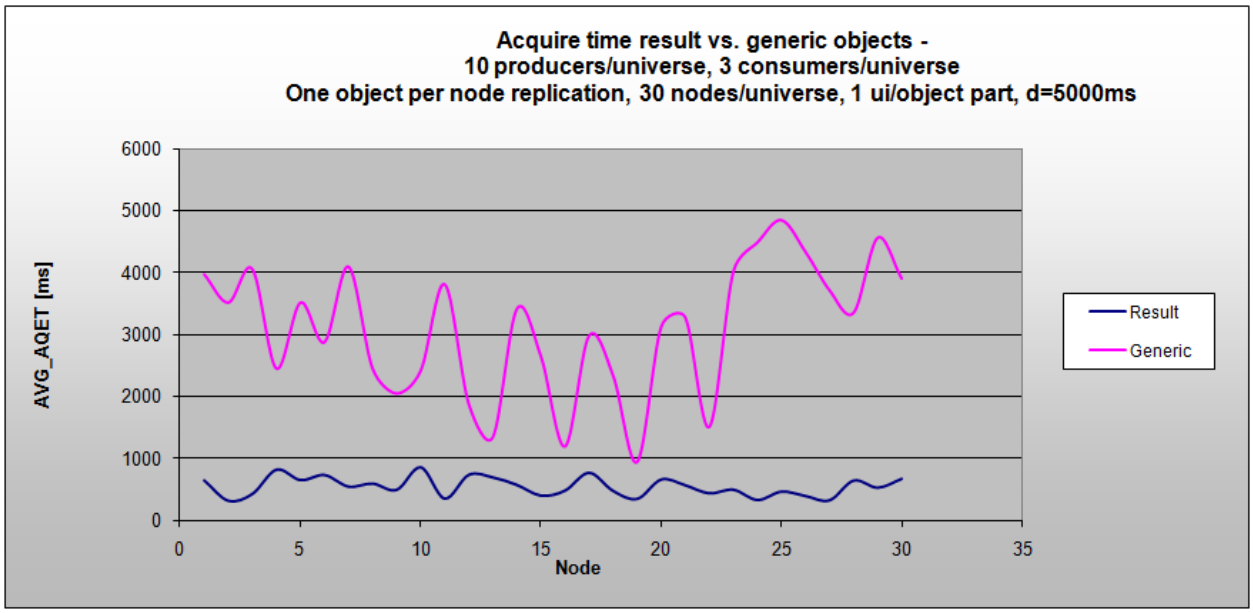

Figure 106: Acquire exclusive time: result vs. generic objects for 10:3 producers: consumers

Parameter	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]	CT [ms]
result	1402	84	1508	29	2100257
generic	448	968	710	68	2100156

Table 32: Result vs. generic objects for 10:3 producers: consumers

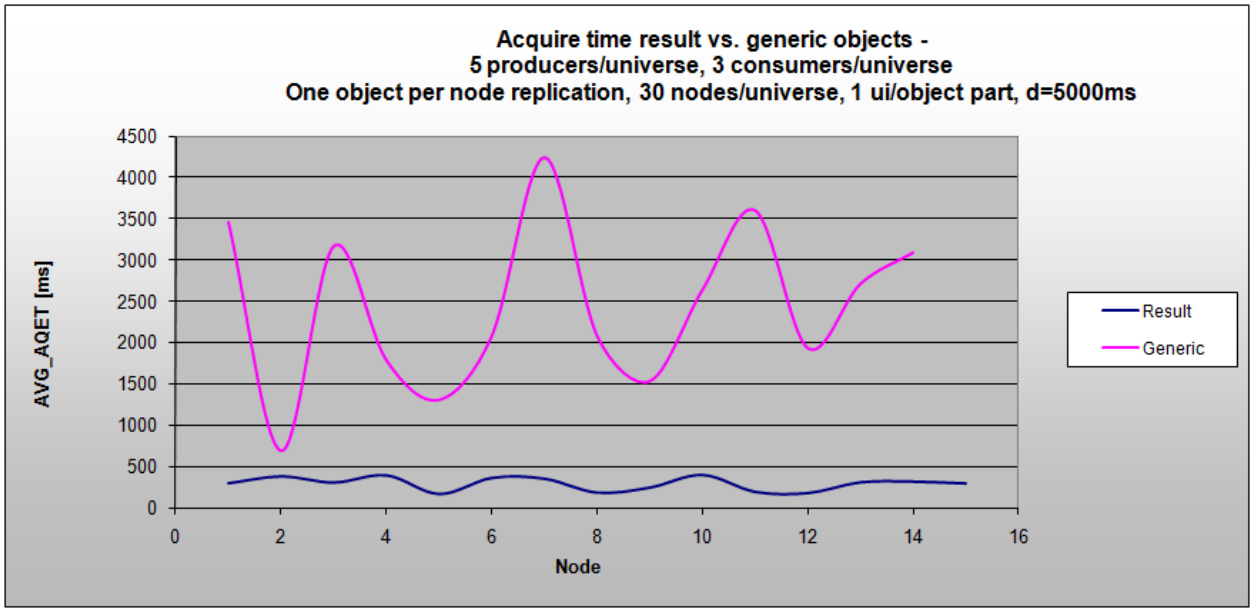

Figure 107: Acquire exclusive time result vs. generic objects for 5:3 producers: consumers

Increasing the producer-consumer ratio from 10:3 to 5:3 and keeping the same experiment parameters as in the previous case, it can be noticed that the result objects have an increasing acquire

exclusive time, but lower values for the other parameters since the number of consumers have reduced and thus the overall update penalty. Compared to case of generic objects, result object show an overall better performance.

Object type	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]	CT [ms]
result	761	100	871	31	2100280
generic	443	718	730	68	2100302

Table 33: Result vs. generic objects for 10:3 producers: consumers

By changing the frequency of the acquire requests using 2000ms, 3000ms and 5000ms delays and keeping the other parameters unchanged, the results presented in Figure 108 and Table 34 show the evolution path towards an unstable system at d=2000ms, where there is a dramatic increase in both acquire exclusive and acquire time, while the object synchronization time remained approximately the same.

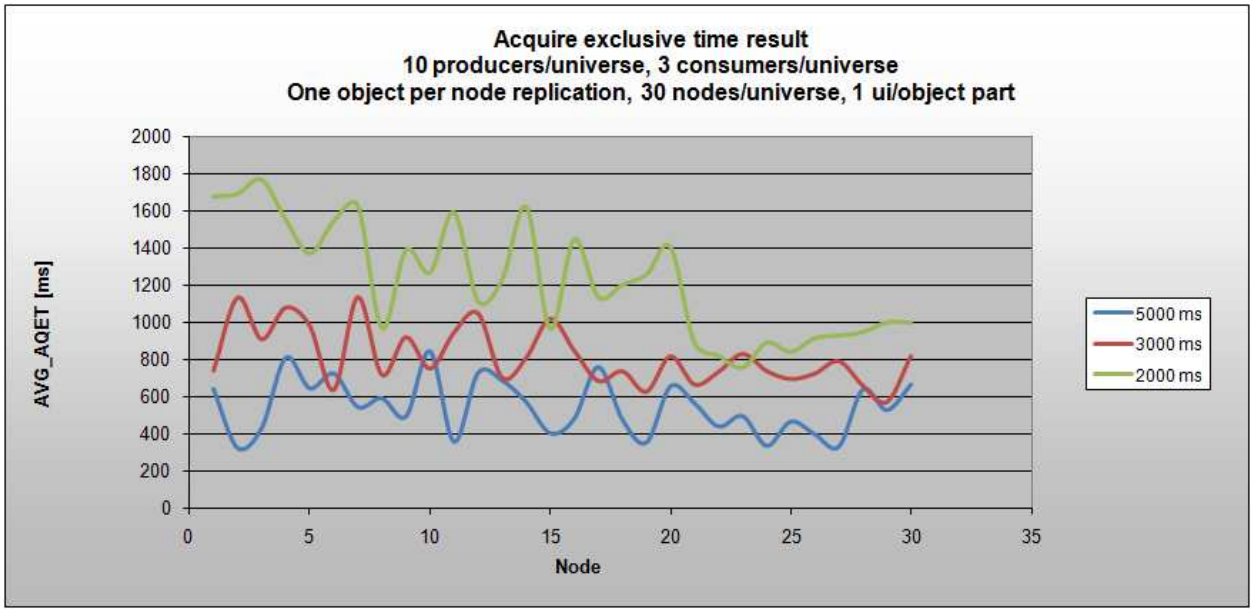

Figure 108: Acquire exclusive time variation with d for result objects

d	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]	CT [ms]
5000 ms	1402	84	1508	29	2100257
3000 ms	1400	116	1495	29	1300287
2000 ms	1394	534	2657	30	900259

Table 34: Parameter variation with d for result objects

The experimental results obtained by changing the producer-consumer ratio in different system configurations are shown in Figure 109, Figure 110 and Table 35. While in the first case where d=5000ms, there was no major difference in terms of the system parameters (e.g. stable system), regardless of the producer-consumer ratio, in case the delay was decreased to 2000ms, it can be noticed that as the consumer-producer ratio increases, the acquire exclusive increases too. This behavior is the expected one since the system is processing an increasing number of operations per second and the primary node's queues become saturated.

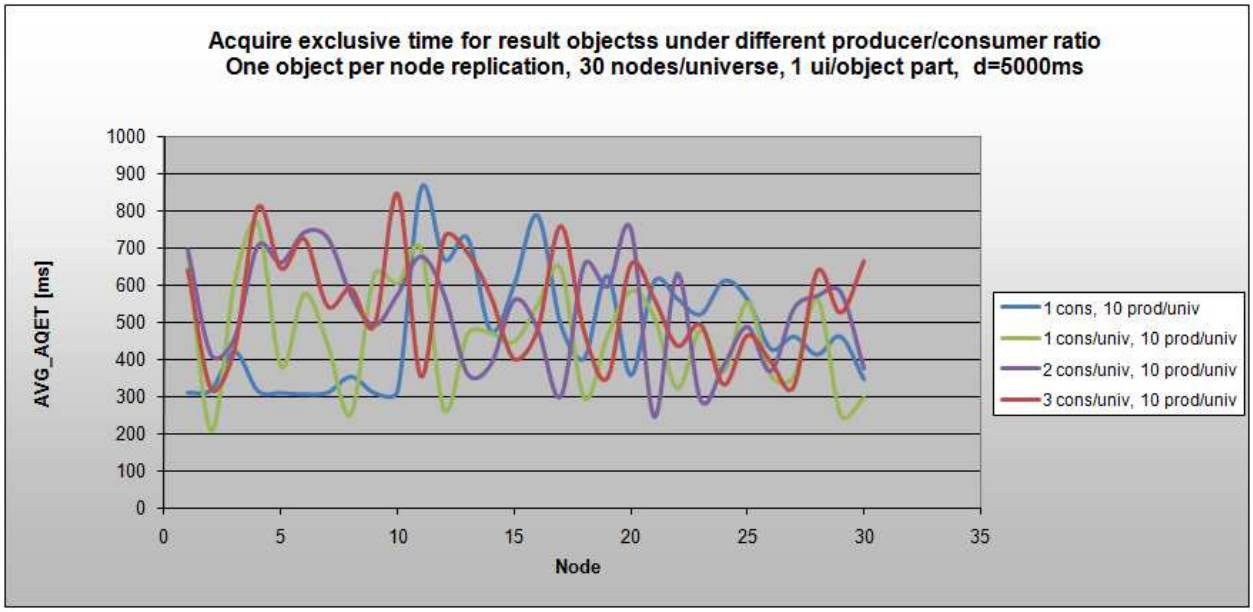

Figure 109: Acquire exclusive time for result objects at different p-c rates, d=5000ms

Variant	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]
1 cons, 10 prod/univ	1454	0	1965	37
1 cons/univ, 10 prod/univ	1414	80	1545	30
2 cons/univ, 10 prod/univ	1392	84	1989	32
3 cons/univ, 10 prod/univ	1402	84	1519	30

Table 35: Parameter variation for result objects at different p-c rates

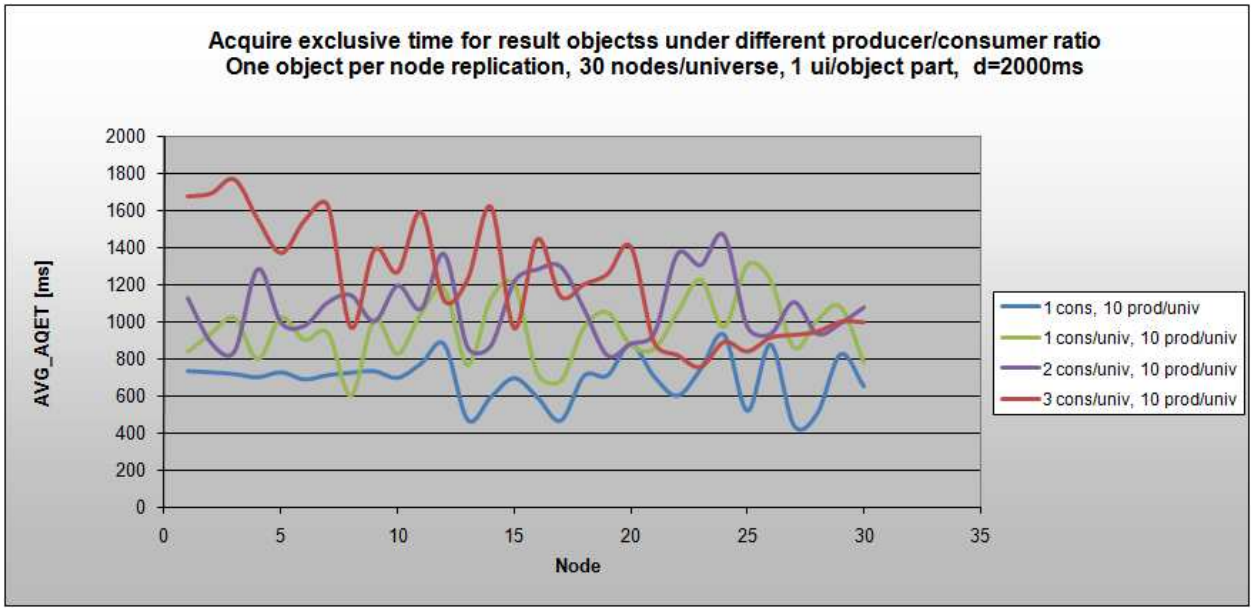

Figure 110: Acquire exclusive time for result objects at different p-c rates, d=2000ms

7.2.10 Grid Write-Mostly Objects

The experiments for result objects described in Chapter 6 specified a number of nodes up to 50 nodes per universe. Similar to the previous cases, due to physical limitation in available computing cores the experiment was conducted in a configuration of up to 30 nodes per universe and a number of 3 universes.

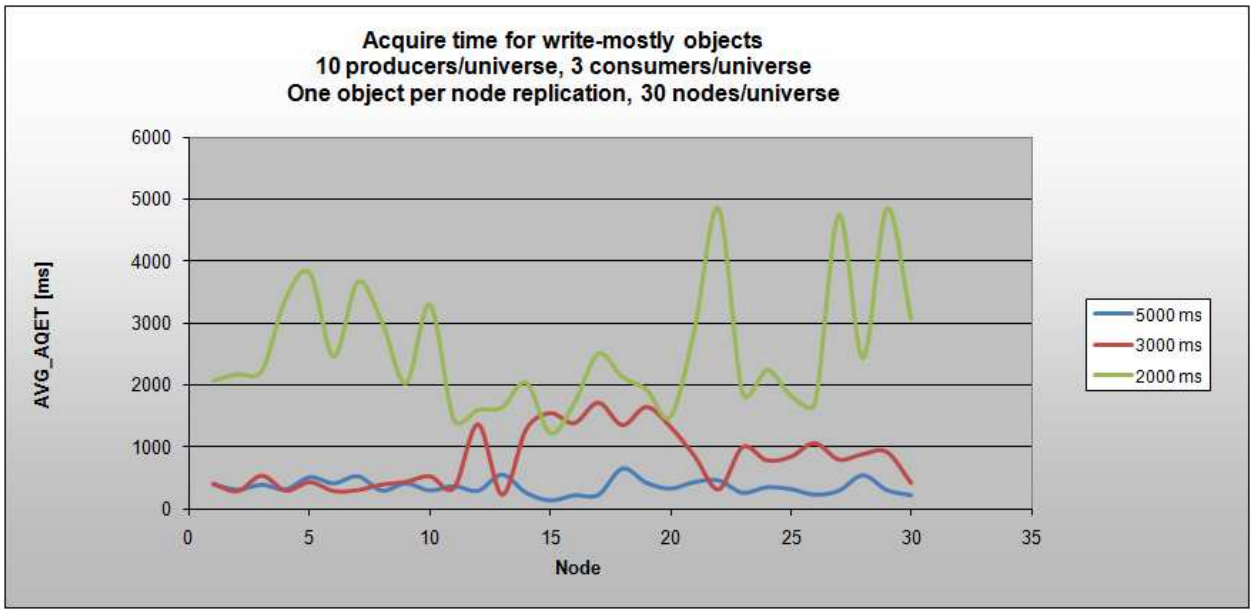

Figure 111: Acquire exclusive time for write mostly and different acquire frequencies

Considering a producer-consumer ratio of 10:3, one object per node replication rule and 30 nodes per universe and changing the operation frequency by choosing d=2000ms, d=3000ms and d=5000ms, the results illustrated in Figure 111 and Table 36 were obtained. It can be noticed that by increasing the operation frequencies, both AQET and AQT parameters are increasing their values, indicating a saturation of the primary node's queues and the path towards an unstable system.

d	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]
5000 ms	146	95	268	34
3000 ms	145	397	286	35
2000 ms	138	324	1441	34

Table 36: Parameter values for write mostly and different acquire frequencies

For one of the selected configurations, by selecting different replication rules like "one object per universe" and "one object per node", according to Figure 112 and Table 37, it can noticed that by having one replica per universe the AQET is reduced to the same values as in case of a stable system (e.g. *d*=5000ms). The same reduction can be seen for the AQT parameter, corresponding to a stable system too. This means that starting with a certain producer-consumer ratio; it is more efficient to limit the number of replicated objects, since a higher number of replicas generate increasing update penalties.

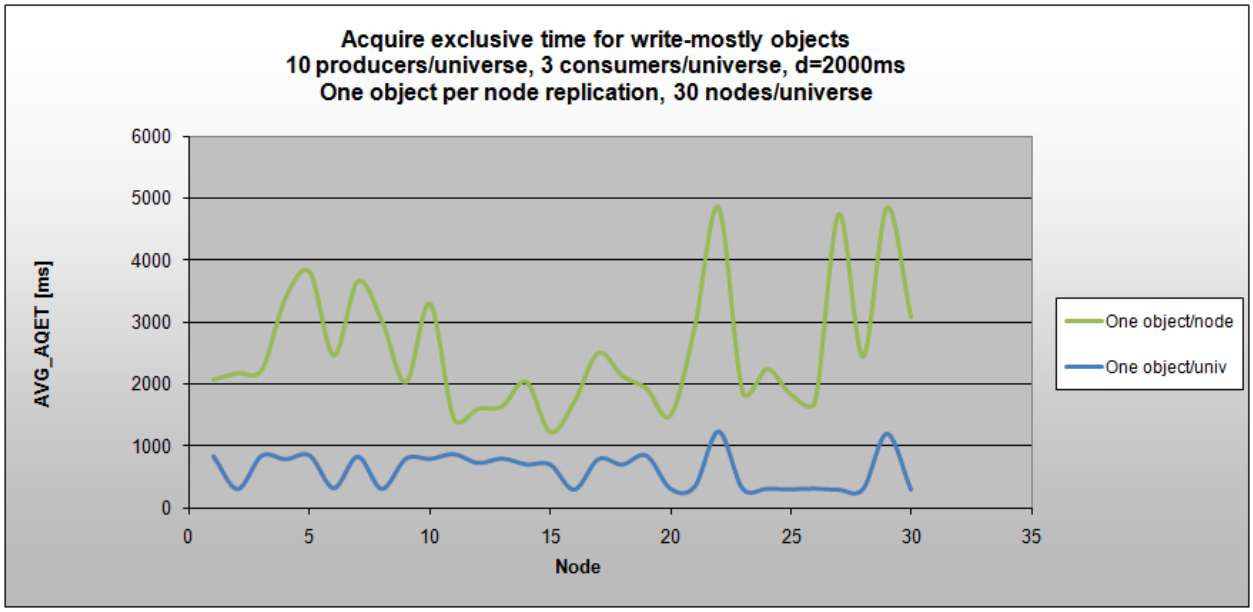

Figure 112: Acquire exclusive time dependency to the replication rule

Variant	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]
One object/node	138	324	1441	34
One object/universe	170	455	307	38

Table 37: Parameter dependency to the replication rule

For two different configurations where *d*=2000ms and *d*=5000ms, by changing the producer-consumer ratio, the results summarized in Figure 113, Table 38, respectively Figure 114 and Table 39 were obtained. Similar to the case of result objects, the first situation shows a stable system where all parameters are relatively constant, while the second experiment shows increasing AQET and AQT times and higher variations, denoting the path towards an unstable system.

Variant	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]
1 cons, 10 prod/univ	132	70	180	28
1 cons/univ, 10 prod/univ	175	37	284	31
2 cons/univ, 10 prod/univ	157	41	285	32
3 cons/univ, 10 prod/univ	146	89	275	30

Table 38: Parameter variation for different p:c ratios and d=5000ms

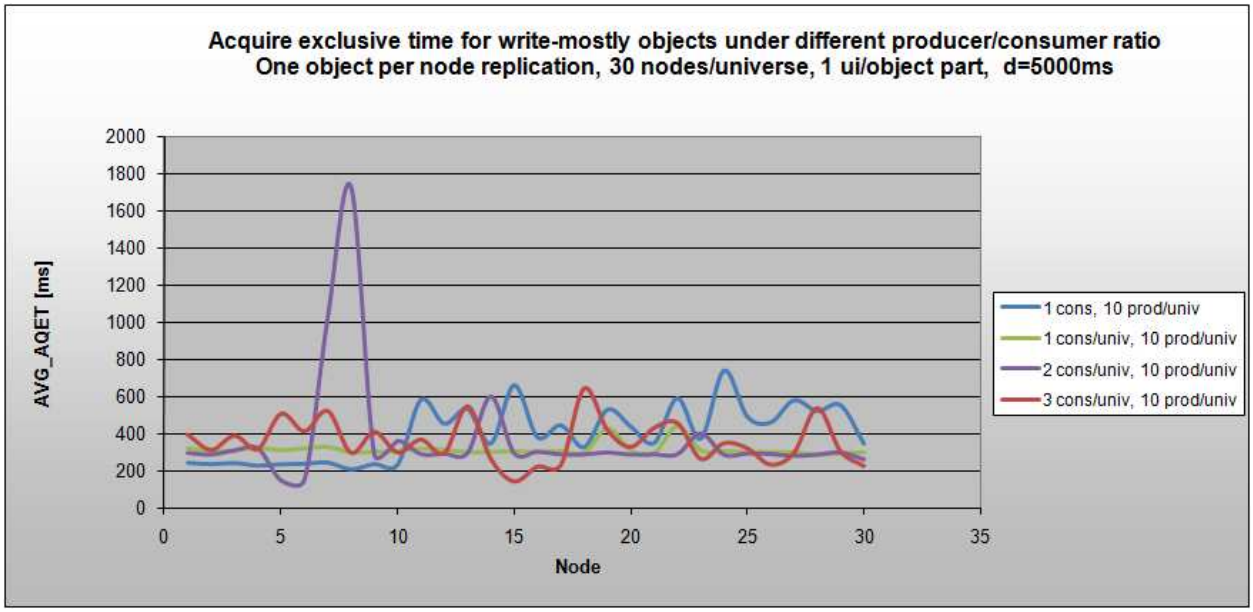

Figure 113: Acquire exclusive time variation for different p:c ratios and d=5000ms

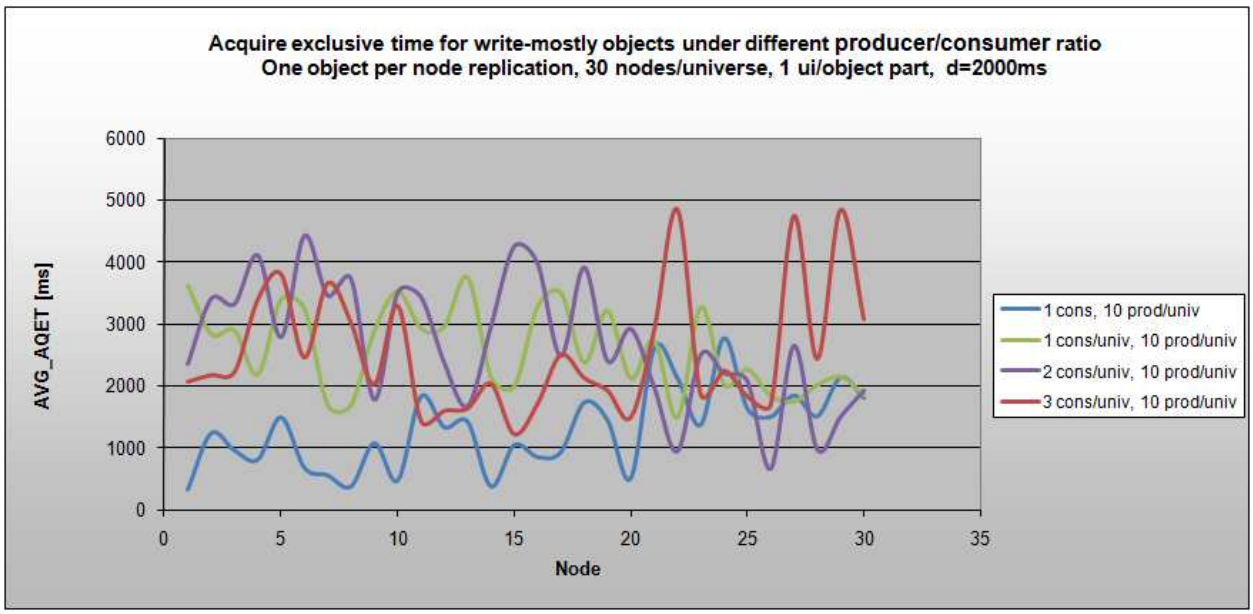

Figure 114: Acquire exclusive time variation for different p:c ratios and d=2000ms

Variant	OST [ms]	TT [ms]	AQT [ms]	RLT [ms]
1 cons, 10 prod/univ	145	151	1226	29
1 cons/univ, 10 prod/univ	153	237	2688	36
2 cons/univ, 10 prod/univ	145	268	3503	37
3 cons/univ, 10 prod/univ	138	324	4230	36

Table 39: Parameter variation for different p:c ratios and d=2000ms

8 Conclusions

The evolution of distributed systems from the conceptual and software technologies point of view, introduced more and more possibilities in programming solutions. One of the many programming paradigms in distributed systems, and also the focus of this thesis is shared data programming. While being considered outside of the common programming solutions, mainly dominated by message passing concept, we believe that shared data programming provides an attractive alternative to MPI-like solutions where the programmer does not have to explicitly orchestrate data exchange between processes, but rather focus on the algorithmic complexities of the problem to be solved. In order to support this idea we had, based on carefully monitoring the grid domain for some time, we started to analyze the drawbacks of existing solutions followed by a model design and prototype implementation of a novel solution which aims to respond to the call for an alternative grid programming concept.

The first part of the thesis focused on a critical analysis of grid programming concepts and technologies with a focus on the shared data programming paradigm which are barely visible in today's grid programming solutions. Distributed shared programming concepts were considered as a viable and interesting approach especially in highly interactive applications where message passing would increase application's complexities and decrease programmability and potentially performance. The dilemma of distributed shared data programming was analyzed from several perspectives and the major qualities of a design were highlighted.

In the second part of the thesis we have defined a model for distributed shared data programming based on the concept of universes and grid nodes. Out of several potential architectural solutions we have selected the most promising one that shall represent a good tradeoff between complexity, scalability and performance. The overall architecture has been augmented by adding a coherence type dimension where specialized objects are considered for certain interaction patterns.

The final part of the thesis addressed the analysis of both the model as well as the prototype implementation of the grid service layer for shared data programming. While the theoretical analysis focused on a static system where no message queues have been modeled, the prototype analysis exposes the behavior of an implementation where all aspects found in real deployments are present.

One of the first observations during the experiments was that it was very easy and straightforward to program a distributed application using the prototype implementation of the universe model (GUN). GUN does not require any knowledge of MPI-like programming concepts, and it requires only the algorithmic representation of the problem. It was easier and more convenient to express interactions via shared data rather than messages. Additionally, it was very straightforward to orchestrate data exchange via workers since there is a built-in data representation at the grid shared object. Even if this is a subjective remark, it appeared quite easy to define distributed data abstractions and let their life cycle be managed by GUN automatically. From the deployment point of view, GUN was run in different heterogeneous configurations, spanning its universes across Europe and Central Asia. In general deployment aspects are more difficult to manage, but this aspect is an inherent issue of distributed computing and especially grids.

The concepts defined in this work as well as the GUN prototype represent an attractive MPI-like alternative to program large scale grid applications following the shared data paradigm. The solution makes use of a flexible deployment model based on the network's characteristics as well as a

customizable object replication configuration. Having a good scalability and built-in object life-cycle management, it provides an easy to use concept for shared data programming on the grid. Considering the experimental results, the GUN prototype showed a good performance as well as scalability. From the object usage point of view, the following recommendations can be made:

- Use read-only objects for immutable objects as much as possible
- Use migratory objects only when a very high method invocation penalty cannot be avoided
- Use private objects for interactions within a universe or where the number of writers is very low
- Use producer-consumer objects where there is only one writer and multiple readers across universe (otherwise use private objects)
- Use result objects where the object state can be composed in non-conflicting parts that are written frequently and independently

The main contribution of this thesis consists of a novel solution to shared data programming for large scale distributed systems. The solution, together with this work brings the following original contributions in the context of grid computing domain:

- A comprehensive overview on grid computing landscape, including its history, with a focus on grid programming paradigms. Contrary to most synthesis papers on grid computing which concentrate on grid infrastructure, grid middleware or grid applications, we focused our attention on grid programming paradigms and the challenges in constructing grid applications based on different programming paradigms and concepts.
- Design of a novel model to support distributed shared data programming on the grid where both relaxed consistency is combined with type coherence in an object oriented fashion.
- Detailed design and implementation of a flexible and scalable architecture for a grid service layer for shared data programming which implements the defined model. The solution's novelty comes out of the combination of several design decisions: entry consistency specification, type coherence, three algorithmic extensions to existing mutual exclusion algorithms, and an easy to use object oriented interface to the application programmer.
- Propose a methodology to evaluate the performance and qualitative aspects of shared data programming solutions by defining a clear set of experiments. Contrary to application specific evaluation, the experiments define a clear set of interaction patterns which are easy to orchestrate and implement.
- Design and implement a Java based prototype solution (GUN) which implements the defined architecture using standard functionalities of the Java platform for increased portability.
- Validate the model, architecture and prototype implementation by conducting both theoretical analysis of the model as well as a prototype-based performance benchmarking. The analysis provides a practical evaluation, in both real and simulated environments, and a large set of detailed performance results, which confirm and demonstrate the feasibility of the original ideas developed in the context of this work.

The next steps which we have already undertaken relate to the model analysis through formal methods using a probabilistic model checker which aims to complement the theoretical and prototype based evaluation and support early model analysis and evaluation. At the time of this writing a formal model based on the PRISM model checker has been developed, based on which we could formally check the correctness of the model. During the preliminary activities towards the quantitative performance analysis we reached a state explosion state while increasing the number of universes and their nodes. Consequently the model has to be re-factored in order to overcome the state explosion problem and to be able to execute the complete set of experiments. Last but not least, we aim to apply the developed concept to other engineering domains which could benefit from the shared data programming model. One of these domains relate to virtualization techniques for building next generation multi-core mobile communication systems like the ones developed within the eMuCo project (www.emuco.eu).

Bibliography

[1] Anath Grama, Anshul Gupta, George Karypis, Vipin Kumar. *Introduction to Parallel Computing, Second Edition.* Addison-Wesley, January 2003.

[2] George Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems Concepts and Design, Fourth Edition.* Addison-Wesley, May 2005

[3] Vladimir Silva. *Grid Computing for Developers*, Charles Rivers Media, December 2005

[4] I. Foster. *What is the grid? A Three Point checklist.* Argonne National Laboratory & University of Chicago, July 2002

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, November 1998

[6] Heinz Stockinger: Defining the grid: a snapshot on the current view. The Journal of Supercomputing 42(1): 3-17 (2007)

[7] I. Foster, C. Kesselman, and S. Tuecke. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". International J. Supercomputer Applications 15(3), 2001.

[8] The Search for Extraterrestrial Intelligence, SETI@home, http://setiathome.ssl.berkeley.edu

[9] FAFNER, http://www.npac.syr.edu/factoring.html

[10] Foster, I., Geisler, J., Nickless, W., Smith, W. and Tuecke, S. (1997) Software infrastructure for the I-WAY high performance distributed computing experiment. Proc. 5th IEEE Symposium on High Performance Distributed Computing, 1997, pp. 562–571.

[11] Globus Toolkit Version 4: Software for Service-Oriented Systems. I. Foster. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.

[12] Fran Berman, Geoffery Fox, Tony Hey. *Grid Computing, Making the Global Infrastructure a Reality.* John Wiley & Sons, October 2003.

[13] Baker, M. Smith, G, Jini meets the Grid, Proceedings of the International Conference on Parallel Processing Workshops, pp. 193-198, ISBN: 0-7695-1260-7, Spain 2001.

[14] Condor Scheduler, http://www.cs.wisc.edu/condor/.

[15] Sun Grid Engine, http://wwws.sun.com/software/gridware/.

[16] The Storage Resource Broker, http://www.npaci.edu/DICE/SRB/.

[17] Buyya, R., Abramson, D. and Giddy, J. (2000) Nimrod/G: An architecture for a resource management and scheduling system in a global computational Grid. The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China, 2000.

[18] HotPage, https://hotpage.npaci.edu

[19] SDSC GridPort Toolkit, http://gridport.npaci.edu/.

[20] NLANR Grid Portal Development Kit, http://dast.nlanr.net/Features/GridPortal/.

[21] Allen, G., Dramlitsch, T., Foster, I., Karonis, N., Ripeanu, M., Seidel, Ed. and Toonen, B. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus, Winning Paper for Gordon Bell Prize (Special Category), Supercomputing 2001, August, 2001; Revised version.

[22] Almond, J. and Snelling, D. (1999) UNICORE: Uniform access to supercomputing as an element of electronic commerce. Future Generation Computer Systems, 15, 539–548.

[23] The DataGrid Project, http://eu-datagrid.Web.cern.ch/.

[24] Akarsu, E., Fox, G. C., Furmanski, W. and Haupt, T. WebFlow: High-level programming environment and visual authoring toolkit for high performance distributed computing. SC98: High Performance Networking and Computing, Orlando, FL, 1998.

[25] Mitra, N. (2001) SOAP Version 1.2 Part 0: Primer, W3C Working Draft, December 17, 2001, www.w3.org/TR/2001/WD-soap12-part0-20011217/.

[26] Web Services Description Language (WSDL) Version 1.1, W3C Note 15, March, 2001, http://www.w3.org/TR/wsdl.

[27] UDDI: Universal Description, Discovery and Integration, http://www.uddi.org.

[28] Leymann, F. (2001) The Web Services Flow Language, http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May, 2001.

[29] Foster, I. et al. (2002) The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Argonne, IL: Argonne National Laboratory, Available at http://www.globus.org/research/papers/ogsa.pdf.

[30] Global Grid Forum Web Site, http://www.gridforum.org.

[31] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire. From Open Grid Services Infrastructure to WSResource Framerowk: Refactoring & Evolution. Available online at http://www.chinagrid.net/dvnews/upload/2005_04/05040200359561.pdf

[32] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: A Large Scale, Recon_gurable, Controlable and Monitorable Grid Platform. In Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05), Seattle, Washington, USA, November 2005.

[33] TeraGrid Project, http://www.teragrid.org, August, 2002.

[34] NASA Information Power Grid, http://www.ipg.nasa.gov/.

[35] DOE Science Grid, http://www.doesciencegrid.org.

[36] Taylor, J. M. and e-Science, http://www.e-science.clrc.ac.uk and http://www.escience-grid.org.uk/

[37] Protein Data BankWorldwide Repository for the Processing and Distribution of 3-D Biological Macromolecular Structure Data, http://www.rcsb.org/pdb/.

[38] MyGrid – Directly Supporting the e-Scientist, http://www.mygrid.info/.

[39] Biomedical Informatics Research Network BIRN Grid, http://www.nbirn.net/.

[40] GEODISE Grid for Engineering Design Search and Optimization Involving Fluid Dynamics, http://www.geodise.org/.

[41] Distributed Aircraft Maintenance Environment DAME, http://www.cs.york.ac.uk/dame/

[42] GriPhyN (Grid Physics Network), http://www.griphyn.org.

[43] Particle Physics Data Grid (PPDG), http://www.ppdg.net/.

[44] D. Bosio and J. Casey and A. Frohner and L. Guy, Next generation EU datagrid data management services, Computing in high energy physics, (CHEP2003), 2003.

[45] GridCoord Initiative Report on Survey of Funding Bodies and Industry, revision v.2.2, released on 22.11.2005, http://gridcoord.org

[46] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz, “A Grid Programming Primer,” Tech report, Advanced Programming Models Research Group, August 2001

[47] Nakada, H., Matsuoka, S., Seymour, K., Dongarra, J., Lee, C. and Casanova, H. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. 3rd International Workshop on Grid Computing, LNCS, Vol. 2536, November, 2002, pp. 274–278.

[48] Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U. and Takagi, H. (1997) Ninf: a network based information library for a global world-wide computing infrastructure. Proc. of HPCN ’97 (LNCS-1225), 1997, pp. 491–502.

[49] Casanova, H. and Dongarra, J. (1996) NetSolve: a network server for solving computational science problems. Proceedings of Super Computing ’96, 1996.

[50] Martin Alt and Sergei Gorlatch, Adapting Java RMI for grid computing, Future Generation Computing Systems, Vol. 21, No. 5, Pp. 699-707, 2004

[51] A. Rowston, P. Druschel*: Pastry. Scalable, distributed object location and routing for large-scale peer-to-peer systems.* In Proc. IFIP/ACM Middleware, Heidelberg, Germaniy, Nov. 2001

[52] B. Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz. *Tapestry: A Resilient Global-Scale Overlay for Service Deployment*, IEEE Journal on Selected Areas in Communications, Vol. 22, No. 1, January 2004

[53] I. Stoica, R. Morris, R. Karger, D. Haashoek, H. Balakrishnan. *Chord: A scalable Peer-to-Peer lookup service for internet applications.* In ACM SIGCOMM, August 2001

[54] Gong, L. JXTA: A Network Programming Environment. IEEE Internet Computing, 5(3), Pp. 88–95, 2001.

[55] Rana, O. F., Getov, V. S., Sharakan, E., Newhouse, S. and Allan, R. Building Grid Services with Jini and JXTA, February, 2002, GGF2 Working Document.

[56] W. M. P. van der Aalst. Don't go with the flow: Web Services composition standard exposed. IEEE Intelligent Systems, Jan/Feb 2003.

[57] T. Andrews et al, BPEL: Business Process Execution Language for Web Services, available at http://www.ibm.com/developerworks/library/specification/ws-bpel/

[58] Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. *GSFL: A Workflow Framework for Grid Services*. Argonne National Laboratory, Preprint ANL/MCS-P980-0802, Aug 2002.

[59] F. Leymann, Web Services Flow Language (WSFL) specification, available at http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[60] S. Thatte, XLANG: Web services for business process design, available at http://www.ebpml.org/xlang.htm

[61] Assaf Arkin et al, WSCI: Web Services Choreography Interface, available at http://www.w3.org/TR/wsci/, 2002

[62] Parallel Computing Message Passing Interface, http://www.mpi-forum.org/.

[63] Foster, I. and Karonis, N. T. (1998) A grid-enabled MPI: message passing in heterogeneous distributed computing systems. Supercomputing. IEEE, November, 1998, www.supercomp.org/sc98.

[64] MPI-2: Extensions to the Message-Passing Interface, Technical Report, University of Tennessee, Knoxville, 1996.

[65] Martin Pinzger, Johann Oberleitner, Harald Gall, "Analyzing and Understanding Architectural Characteristics of COM+ Components," *iwpc*, p. 54, 11th IEEE International Workshop on Program Comprehension (IWPC'03), 2003

[66] N. Carriero and D. Gelernter. Linda in Context. Comms. ACM, Vol. 32, No. 4, pp. 444-58, 1989

[67] E. Freeman, S. Hupfer, K. Arnold. *JavaSpaces: Principles, Patterns and Practice.* Addison-Wesley, New York, 1999

[68] P. Wyckoff, S. McLaughry, S. Lehman and D. Ford. TSpaces. IBM Systems Journal, Vol. 37, No. 3, 1998

[69] Nicholas Carriero, David Gelernter. *A Computational Model for Everything.* CACM 44(11): 77-81, 2001

[70] V. S. Sunderam, PVM: a framework for parallel distributed computing, Journal on Concurrency, Practice and Experience, Volume 2, Number 4, John Wiley & Sons, Chichester, West Sussix, Pages 315-340, 1990.

[71] Joe Throop, OpenMP: Shared-Memory Parallelism From the Ashes, IEEE Computer Society, vol. 32, no. 5, pp. 108-109, May, 1999.

[72] Ken Kennedy, Charles Koelbel and Hans Zima. *The rise and fall of High Performance Fortran: an historical object lesson,* Proceedings of the third ACM SIGPLAN conference on History of programming, Pages: 7-1 - 7-22, ISBN:978-1-59593-766-X, San Diego, California, 2007

[73] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In G. V. Wilson and P. Lu, editors, Parallel Programming in C++, pages 547–588. MIT Press,1996.

[74] Beckman, P. Gannon, D. Johnson, E, Portable parallel programming in HPC++, Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing, pp.132-139, ISBN: 0-8186-7623-X, 12 August 1996.

[75] Borja Sotomayor, Lisa Childers, *Globus Toolkit 4, : Programming Java Services,* Morgan Kaufmann, 1st edition, ISBN-10: 0123694043, December, 2005.

[76] Karasavvas, Konstantinos and Antonioletti, Mario and Atkinson, Malcolm and Hong, Neil C. and Sugden, Tom and Hume, Alastair and Jackson, Mike and Krause, Amrey and Palansuriya, Charaka, Introduction to OGSA-DAI Services, Lecture Notes in Computer Science, pp. 1-12, Volume 3458, 2005

[77] gLite – Lightweight Middleware for Grid Computing http://cern.ch/glite/

[78] Justin Bradley, Christopher Brown et. al., The OMII Software Distribution, Proceedings of the UK e-Science All Hands Meeting. 2006, Nottingham UK, 2006, pp. 748–753.

[79] Allcock, W., Chervenak, A., Foster, I., Kesselman, C. and Tuecke, S. Protocols and services for distributed data-intensive science. Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT), pp. 161–163, 2000

[80] Cheung, B.W.L. Cho-Li Wang Lau, F.C.M. LOTS: a software DSM supporting large object space, IEEE International Conference on Cluster Computing, Pp. 225-234, ISBN: 0-7803-8694-9, 2004

[81] J.P. Ryan, B.A. Coghlan, SMG: Shared memory for Grids, In: Proceedings of 6th IASTED International Conference on Parallel and Distributed Computing and Systems. 2004, pp. 439-451. http://www.cs.tcd.ie/coghlan/pubs/pdcs04-06072004-v1.pdf

[82] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, Ce-Kuen Shieh, Teamster-G: a grid-enabled software DSM system, Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005, Volume 2, pp. 905- 912, ISBN: 0-7803-9074-1, May 2005.

[83] Gabriel Antoniu, Luc Bouge, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. Scalable Computing: Practice and Experience, 6(3):45-55, November 2005.

[84] J. Osrael, L. Froihofer, and K.M. Goeschka. A Replication Model for Trading Data Integrity against Availabilit, The 12th Int. Symp. on Pacific Rim Dependable Computing (PRDC'2006), IEEE CS Press, 2006

[85] Leach, P. J., Levine, P.H., Douros, B.P. et al. *The architecture of an integrated local networkt.* IEEE J. Selected Area in Communications, Vol SAC-1, No 5, pp. 842-56

[86] John B. Carter, Dilip Khandekar, Linus Kamb. *Distributed Shared Memory: Where We Are and Where We Should Be Headed.* Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), Page: 119, IEEE Computer Society, ISBN:0-8186-7081-9, 1995

[87] D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J, Hennesy, M. Horowitz, M.S. Lam. *The Stanford Dash multiprocessor.* IEEE Computer, Vol. 25, No. 3, pp. 63-79, 1992

[88] R. Bisiani, M. Ravishankar. *Plus: a distributed shared memory system.* In Proceedings 17th International Symposium on Computer Architecture, pp. 115-24, 1990

[89] K. Li, P. Hudak,. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, Vol. 7, No. 4, pp. 321-59, 1989

[90] J.B. Carter, J.K. Bennett and W. Zwaepepoel. *Implementation and performance of Munin*. In Proceedings 13th ACM Symposium on Operating System Principles, pp. 152-64, 1991

[91] P. Dasgupta, R.J. LeBlanc Jr, M. Ahamad and U. Ramachandran. *The Clouds distributed operating system*. IEEE Computer, Vol. 24, No. 11, pp. 33-34, 1991

[92] B. Fleisch and G. Popek. *Mirage, a coherent distributed shared memory design*. In Proceedings 12th ACM Symposium on Operating System Principles, December 1989, pp. 211-23.

[93] Heri E. Bal, M. Frans Kaashoek and Andrew S. Tanenbaum, Orca: a language for parallel programming of distributed systems", Proceedings of the IEEE Transactions on Software Engineering, Volume 18, Number 3, Pp. 190--205, 1992.

[94] M. Frans Kaashoek and Andrew S. Tanenbaum, An Evaluation of the Amoeba Group Communication System. International Conference on Distributed Computing Systems, pp. 436-448, 1996

[95] R.E. Kessler and M. Livny. An Analysis of Distributed Shared Memory Algorithms, In Proceedings 9th International Conference Distributed Computing Systems, IEEE, pp. 98-104, 1989

[96] Pradeep K. Sinha. *Distributed Operating Systems Concepts and Design*, IEEE Press, Prentice Hall of India, New Delhi, 2005

[97] Y. Saito, M. Shapiro. *Optimistic replication*, Technical report, MSR-TR-2003-60, September 2003

[98] M. Wiesmann, F. Perdonne, A. Schiper, B. Kemmer, G. Alonso.. *Understanding replication in databases and distributed systems*, In Proceeedings of the 20th International Conference on Distibuted Computing Systems (ICDCS 2000), pp 264-274, IEEE, April 2000

[99] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. IEEE Transactions Computers, Vol. C-28, No. 9, pp. 690-1

[100] P. Hutto and M. Ahmad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In proceedings 10th International Conference on Distributed Computer Systems, IEEE, pp. 302-11, 1990

[101] R. Lipton, J. Sandberg. *PRAM: a scallable shared memory.* Technical Report CS-TR-180-88, Princeton University, 1988

[102] J. Goodman, Cache Consistency and Sequencial Consistency. Technical Report 61, SCI Committee, 1989

[103] M. Dubois, C. Scheurich, F.A. Briggs. *Synchronization, coherence and event ordering in multiprocessors..* IEEE Computer, Vol. 21, No. 2, pp 9-21, 1988

[104] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy . Memory Consistency and Event Ordering in Scalable Shared.Memory Multiprocessors. In Proceedings 17th Annual International Symposium on Computer Architecture, May 1990, pp. 15-26

[105] P. Keleher, A. Cox and W. Zwaenepoel. Lazy Consistency for software distributed shared memory. In Proceedings 19th Annual International Symposium on Computer Architecture. Pp. 13-21, May 1992

[106] B. Bershad, M. Zekauskas, W. Sawdon *The Midway distributed shared memory system.* In Proceedings IEEE COMPCON Conference, IEEE, pp 528-37, 1993

[107] L. Iftode, J. Singh, K. Li. *Scope consistency: a bridge between release consistency.* In Proceedings 8th annual ACM symposion on Parallel algorithms and architectures. pp. 277-87, 1996

[108] Huang, Z. Y. Purvis, M. Werstein, P. View-Oriented Parallel Programming and View-Based Consistency. Lecture notes in computer science, 2004, ISSU 3320, pages 505-518, SPRINGER-VERLAG

[109] Wilfried N. Gansterer and Joachim Zottl, Message Passing vs. Virtual Shared Memory. A Performance Comparison

[110] D. Tudor, V. Cretu and H. Ciocarlie. Parallel branch and bound experiment using Java based message passing and shared object space solutions. The 7th International Conference on Technical Informatics, 8-9 June 2006, Timisoara, Romania.

[111] Bershad B.N. and Zekauskas M.J., Midway Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, CMU Report CMU-CS-91-170, Sept. 1991

[112] L. Lamport. Time, clocks, and the ordering of events in a distributed system. CACM, 21(7):558–564, 1978.

[113] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. CACM, 24, 1981.

[114] M. Maekawa. A √N algorithm for mutual exclusion in decentralized systems. ACM-TOCS, 3(2):145 – 159, May 1985.

[115] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. ACM-TOCS, 3(4):344–349, 1985.

[116] K. Raymond. A tree-based algorithm for distributed mutual exclusion. ACM-TOCS, 7(1):61–77, 1989.

[117] M. Naimi, M. Trehel, and A. Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. JPDC, 34(1):1–13, 1996.

[118] S. Beyer, A. Sanchez, F. Munoz, and P. Galdamez, "DeDiSys Lite: An Environment for Evaluating Replication Protocols in Partitionable Distributed Object Systems," Proc. 1st Int. Conf. on Availability, Reliability, and Security, IEEE, 2006, doi:10.1109/ARES.2006.44.

[119] J. Osrael, L. Froihofer, K.M. Goeschka, S. Beyer, P. Galdamez, and F. Munoz, "A System Architecture for Enhanced Availability of Tightly Coupled Distributed Systems," Proc. 1st Int. Conf. on Availability, Reliability, and Security, IEEE, 2006.

[120] I. Chang, M. Singhal, and M. T. Liu, “An improved log (N) mutual exclusion algorithm for distributed systems,” in Proceedings of the 1990 International Conference on Parallel Processing, Aug. 1990, pp. 295–302.

[121] F. Mueller, “Prioritized token-based mutual exclusion for distributed systems,” in Proceedings of 12th Intern. Parallel Proc. Symposium & 9th Symp. on Parallel and Distr. Processing, Mar. 1998, pp. 791–795.

[122] Bertier, M.; Arantes, L.; Sens, P. Hierarchical token based mutual exclusion algorithms. IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004, ISBN: 0-7803-8430-x, Page 539- 546, 19-22 April 2004.

[123] A. Housni and M. Trehel, “Distributed mutual exclusion by groups based on token and permission,” in Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, June 2001, pp. 26–29.

[124] L. Bouge G. Antoniu and S. Lacour, “Making a DSM consistency protocol hierarchy-aware: an efficient synchronization scheme,” in Proceedings of the Workshop on Distributed Shared Memory on Clusters, 2003, pp. 516–521.

[125] Julien Sopena, Fabrice Legond-Aubry, Luciana Arantes, Pierre Sens. A Composition Approach to Mutual Exclusion Algorithms for Grid Applications. Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007), Volume 00, Page: 65, ISBN 0-7695-2933-X, 2007

[126] Colin Blundell, E Christopher Lewis, and Milo Martin. Subtleties of Transactional Memory Atomicity Semantics. In Computer Architecture Letters, Volume 5, Number 2, November 2006

[127] Dan Grossman, Jeremy Manson and William Pugh. What do high-level memory models mean for transactions? Proceedings of the 2006 workshop on Memory system performance and correctness, pp. 62-69, ISBN:1-59593-578-9, San Jose, California, 2006

[128] Dacian Tudor, Vladimir Cretu and Wolfgang Schreiner, "Designing an Architecture for Distributed Shared Data on the Grid", Proceedings of the International Conference on Algorithms and Architectures (ICA3PP-2008), Cyprus, June 9-11, 2008, A. Bourgeois and S.Q. Zheng (Eds.), LNCS 5022, pp. 261–264, Springer.

[129] Baker, M. Smith, G. (2001) ‘Jini meets the Grid’, Proceedings of the International Conference on Parallel Processing Workshops, pp. 193-198, ISBN: 0-7695-1260-7.

[130] Laurent Baduel, Francoise Baude, Denis Caromel. Efficient, Flexible and Typed Group Communications for Java. Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, 2002.

[131] Dacian Tudor, Vladimir Cretu, "Experiences on Grid Shared Data Programming", Proceedings of the Second International Workshop on P2P, Parallel, Grid and Internet Computing (3PGIC-2008), ISBN 0-7695-3109-1, pp. 390-395, Barcelona, Spain, March 4-7, 2008.

[132] Tudor, D., Macariu, G., Schreiner, W. and Cretu, V-I. (2009) 'Experiences on grid shared data programming', International Journal of Grid and Utility Computing, Vol. 1, No. 4, pp.296–307, 2009.

[133] Dacian Tudor, Georgiana Macariu, Wolfgang Schreiner, Vladimir Cretu "Experiments on a Grid Layer Prototype for Shared Data Programming Model", Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Timisoara, Romania, May 28-29, 2009. IEEE Catalog: CFP0945C-CDR, ISBN: 978-1-4244-4478-6.

[134] D. Petcu, D. Gorgan, F. Pop, D. Tudor, D. Zaharie, *"Satellite Image Processing on a Grid-Based Platform"*, International Journal of "Computing", Vol. 7, Issue 2, pp. 51-58, September 2008.

Personal Publications

Tudor, D., Macariu, G., Schreiner, W. and Cretu, V-I. (2009) *'Experiences on grid shared data programming'*, International Journal of Grid and Utility Computing, Vol. 1, No. 4, pp.296–307.

D. Petcu, D. Gorgan, F. Pop, D. Tudor, D. Zaharie, *"Satellite Image Processing on a Grid-Based Platform"*, International Journal of "Computing", Vol. 7, Issue 2, pp. 51-58, September 2008.

Marius Marcu, Dacian Tudor, Horatiu Moldovan, Sebastian Fuicu and Mircea Popa, *"Energy Characterization of Mobile Devices and Applications Using Power-Thermal Benchmarks"*, Microelectronics Journal, doi:10.1016/j.mejo.2008.05.001 , July 2008.

Marius Marcu, Dacian Tudor, Sebastian Fuicu *"Towards a Network-Device Unified Framework for Power-Aware Wireless Applications"*, Proceedings of the 5th International Wireless Communications and Mobile Computing Conference (IWCMC'09), Leipzig, Germany, June 21–24, 2009. ACM, ISBN: 978-1-60558-569-7.

Dacian Tudor, Andrei Stancovici, Bogdan Popescu, Gernot Reisinger, Vladimir Cretu *"Zombee: a Home Automation Prototype for Retrofitted Environments"*, Proceedings of the 2nd International Conference on PErvasive Technologies Related to Assistive Environments (PETRA 2009), Corfu, Greece, June 9-13, 2009. ACM, ISBN: 978-1-60558-409-6.

Dacian Tudor, Georgiana Macariu, Wolfgang Schreiner, Vladimir Cretu *"Experiments on a Grid Layer Prototype for Shared Data Programming Model"*, Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Timisoara, Romania, May 28-29, 2009. IEEE Catalog: CFP0945C-CDR, ISBN: 978-1-4244-4478-6.

Ciprian Stanciu, Dacian Tudor, Vladimir Cretu *"Towards an Adaptable Large Scale Project Execution Monitoring"*, Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Timisoara, Romania, May 28-29, 2009. IEEE Catalog: CFP0945C-CDR, ISBN: 978-1-4244-4478-6.

Dacian Tudor, Georgiana Macariu, Calin Jebelean, Vladimir Cretu *"Towards a Load Balancer Architecture for Multi-Core Mobile Communication Systems"*, Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Timisoara, Romania, May 28-29, 2009. IEEE Catalog: CFP0945C-CDR, ISBN: 978-1-4244-4478-6.

Dacian Tudor, Georgiana Macariu, Calin Jebelean, Vladimir Cretu *"Load Balancing in Modern Multi-Core Mobile Communication Systems"*, Proceedings of 17th International Conference on Control Systems and Computer Science (CSCS 17), Bucharest, Romania, May 26-29, 2009. Vol. 1, pp. 255-261, ISSN: 2066-4451.

Dacian Tudor, Marius Marcu, *"Designing a Power Efficiency Framework for Battery Powered Systems"*, Proceedings of the Israeli Experimental Systems Conference (SYSTOR 2009), Haifa, Israel, May 4-6, 2009. ACM International Conference Proceeding Series ACM 2009, ISBN 978-1-60558-623-6.

Marius Marcu, Dacian Tudor, Sebastian Fuicu, *"A View on Power Efficiency of Multimedia Mobile Applications"*, Proceedings of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 08), Bridgeport, USA, December 5-13, 2008 [Springer listed].

Georgiana Macariu, Dacian Tudor, Vladimir Cretu, *"Designing a Dynamic Replication Engine for Grid Shared Data Programming"*, The 10th International Symposium on Symbolic and Numeric Algorithms for

Scientific Computing, Workshop on Grid Computing Applications Development (SYNASC 2008), pp. 402-409, ISBN 978-0-7695-3523-4, Timisoara, Romania, September 26-29, 2008.

Marius Marcu, Dacian Tudor, Sebastian Fuicu, Mihai Micea, Silvia Copil-Crisan, Florin Maticu, *"Power Characterization of Multi-Threading Mobile Applications"*, Proceedings of the 12th WSEAS Conference on Computers, pp. 583-589, ISBN 978-960-6766-85-5, Heraklion, Crete Island, Greece, July 22-25, 2008.

Dacian Tudor, Vladimir Cretu and Wolfgang Schreiner, *"Designing an Architecture for Distributed Shared Data on the Grid"*, Proceedings of the International Conference on Algorithms and Architectures (ICA3PP-2008), Cyprus, June 9-11, 2008, A. Bourgeois and S.Q. Zheng (Eds.), LNCS 5022, pp. 261-264, Springer-Verlag Berlin Heidelberg 2008.

Marius Marcu, Dacian Tudor, Sebastian Fuicu, Horatiu Moldovan and Voicu Groza, *"A view on mobile terminal power efficiency wireless communication"*, Proceedings of the IEEE International Instrumentation & Measurement Technology Conference (I2MTC-2008), ISBN 1-4244-1541-1, vol. 08, pp. 382, Victoria, Vancouver Island, British Columbia, May 12-15, 2008.

Dacian Tudor and Vladimir Cretu, *"Experiences on Grid Shared Data Programming"*, Proceedings of the Second International Workshop on P2P, Parallel, Grid and Internet Computing (3PGIC-2008), ISBN 0-7695-3109-1, pp. 390-395, Barcelona, Spain, March 4-7, 2008.

Marius Marcu, Dacian Tudor and Sebastian Fuicu, *"Power Efficiency Profile Evaluation for Wireless Communication Applications"*, Proceedings of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 07), Bridgeport, USA, December 3-12, 2007 [Springer listed].

M. Marcu, D. Tudor, H. Moldovan, M. Micea, *"Power Profile Evaluation of Battery-Powered Mobile Applications"*, Proceedings of the 14th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2007), Marrakech, Morocco, December 11-14, 2007, Page(s): 1015 - 1018, ISBN 1-4244-1378-8.

Dacian Tudor, Georgiana Macariu and Vladimir Cretu, *"A Performance Analysis on Message Passing Tools for the Grid"*, Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007), Workshop on Grid Computing Applications Development, IEEE Computer Society Press, Timisoara, Romania, 26-29 Sept 2007, Page(s): 317 - 322, Digital Object Identifier 10.1109/SYNASC.2007.37

Florin Pop, Dacian Tudor, Valentin Cristea and Vladimir Cretu, *"Fault-Tolerant Scheduling Framework for MedioGrid System"*, Proceedings of the IEEE International Conference on ?Computer as a Tool? (EUROCON 2007), Warsaw, Poland 2007, [CD support], IEEE Catalog Number: 07EX1617C, ISBN: 1-4244-0813-X.

Dacian Tudor and Marius Marcu, *"A Power Benchmark Experiment for Battery Powered Devices"*, Proceedings of the 4th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computer Systems: Technology and Applications (IDAACS 2007), [CD support], IEEE Catalog Number: 07EX1838C, ISBN: 1-4244-1348-6, Library of Congress: 2007928159, Dortmund, Germany 2007.

Dana Petcu, Daniela Zaharie, Dorian Gorgan, Florin Pop and Dacian Tudor, "*MedioGrid: a Grid-based Platform for Satellite Image Processing"*, Proceedings of the 4th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computer Systems: Technology and Applications (IDAACS 2007), [CD support], IEEE Catalog Number: 07EX1838C, ISBN: 1-4244-1348-6, Library of Congress: 2007928159, Dortmund, Germany 2007.

Dacian Tudor, Georgiana Macariu and Vladimir Cretu, *"Dynamic Policy Based Replication on the Grid"*, Proceedings of the 4th International Symposium on Applied Computational Intelligence and Informatics (SACI 2007), Timisoara, Romania 2007, pp.77-82, IEEE Catalog Number: 07EX1788, ISBN: 1-4244-1234-X.

Dacian Tudor and Vladimir Cretu, *"A Data Model View on MedioGrid"*, Proceedings of the 16th International Conference on Control Systems and Computer Science (CSCS16), Bucharest, Romania, 22-25 May 2007, pp.150-154, ISBN 978-973-718-743-7

Dacian Tudor, Florin Pop, Valentin Cristea and Vladimir Cretu, *"Towards an IO Intensive Grid Application Instrumentation in MedioGrid"*, Proceedings of the 16th International Conference on Control Systems and Computer Science (CSCS16), Bucharest, Romania, 22-25 May 2007, pp. 130-135, ISBN 978-973-718-741-3

Dacian Tudor, Vladimir Cretu and Horia Ciocarlie, *"A View on Fault Tolerant Techniques Applied to MedioGrid"*, Proceedings of the International Conference on Knowledge Engineering: Principles and Techniques (KEPT 2007), Cluj-Napoca, Romania 2007, ISBN: 978-973-610-556-2, pp.244-251.

D.Tudor, V.Cretu, H.Ciocarlie, *"Parallel Branch and Bound Experiment Using Java Based Messages Passing and Shared Object Space Solutions"*, Proceedings of the 7-th International Conference on Technical Informatics - Conti 2006, Vol.2, Computer and Software Engineering, Timisoara, Romania, 2006, pp. 161-166, ISBN (10) 973-625-321-X, (13) 978-973-625-321-8.

Abbreviations

A - Z	
API	Application Programming Interface
AXIS	Apache Extensible Interaction System
BIRN	Biomedical Informatics Research Network
BPEL	Business Process Execution Language
BPML	Business Process Modeling Language
BPSS	Business Process Specification Schema
CERN	Centre Europenne pour la Recherche Nuclaire
COM	Component Object Model
CPU	Central Processing Unit
DAME	Distributed Aircraft Maintenance Environment
DNS	Domain Name Server
DOE	Department of Energy
DRS	Data Replication Service
DSM	Distributed Shared Memory
EDOC	Enterprise Distributed Object Computing
EGEE	Enabling Grids for E-Science in Europe
FIFO	First-In-First-Out
FTP	File Tranfer Protocol
GGF	Global Grid Forum
GRAM	Grid Resource Allocation and Management
GSFL	Grid Services Flow Language

GSI	Grid Security Infrastructure
HPC	High Performance Computing
IDL	Interface Description Language
IETF	Internet Engineering Task Force
IPC	Inter Process Communication
IPG	Information Power Grid (NASA)
JXTA	Juxtapose (Sun Microsystems)
LAN	Local Area Network
LSF	Load Sharing Facility
LHC	The Large Hadron Collider (CERN)
LRU	Least Recently Used
MDS	Monitoring and Discovery System
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration (USA)
NAT	Network Address Translation
NEES	Network for Earthquake Engineering Simulation (National Science Foundation)
NPACI	National Partnership for Advanced Computational Infrastructure
NSF	National Science Foundation
NUMA	Non-Uniform Memory Architecture
OASIS	Object-Oriented Administrative Systems-development in Incremental Steps
OGF	Open Grid Forum
OGSA	Open Grid Services Architecture
OMII	Open Middleware Infrastructure Institute (UK)

P2P	Peer to Peer
PACI	Partnerships in Advanced Computational Infrastructure
PBS	Portable Batch System
PDL	Process Description Language
POOMA	Parallel Object-Oriented Methods and Applications
PVM	Parallel Virtual Machine
RFT	Reliable File Transfer
RLS	Replica Location Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SETI	Search for Extraterrestrial Intelligence
SMG	Shared Memory for Grids
SOAP	Simple Object Access Protocol (XML protocol)
TCP	Transmission Control Protocol
UDDI	Universal Description and Discovery Interface
VPN	Virtual Private Network
WSCI	Web Service Choreography Interface
WSDL	Web Services Description Language
WSFL	Web Services Flow Language
WSRF	Web Services Resource Framework
XLANG	Web Services for Business Process Design
XML	eXtensible Markup Language
XPDL	XML Processing Description Language

Appendix A

A.1 Read/Write Mutex Java Implementation

```
public class RWLock
{
	private int    givenLocks = 0;
	private int    waitingWriters = 0;
	private Object mutex = new Object();

	public RWLock()
	{
	}

	public void ReadLock()
	{
		synchronized(mutex)
		{
			try
			{
				while((givenLocks == -1) || (waitingWriters != 0))
				{
					mutex.wait();
				}
				catch(java.lang.InterruptedException e){}
				givenLocks++;
			}
		}

	public void WriteLock()
	{
		synchronized(mutex)
		{
			waitingWriters++;
			try
			{
				while(givenLocks != 0)
				{
					mutex.wait();
				}
				catch(java.lang.InterruptedException e){}
				waitingWriters--;
				givenLocks = -1;
			}
		}
	}


	public void ReleaseLock()
	{
		synchronized(mutex)
		{
			if(givenLocks == 0)
				return;
			if(givenLocks == -1)
				givenLocks = 0;
			else
				givenLocks--;

			mutex.notifyAll();
		}
	}
}
```

A.2 Distributed Centralized Algorithm

```
// Every node Ni has following local variables
* self  - keeps the node identification (Ni)
* PN    - keeps the primary gateway identity (e.g. address)

/** Creates a copy of the GridObject o in the Grid Universe and returns a GridObjectRef
       @param  o      The GridObject to be created on the Grid Universe.
       @return        Reference to the created grid object. If operation fails null is returned.
*/
CreateGridObject(GridObject o):
       Call CreateObject(o, self) on PN

/** Finds a GridObject in the Grid Universe that has a given user provided object identifier.
       @param  oid    Object identifier, OID.
       @return        Reference to the found grid object. Returns null if no object could be
found.
*/
FindGridObject(OID oid):
       Call FindGridObject(oid, self) on PN

/** Deletes all GridObjects from the Grid Universe that have a given user provided object
identifier.
       @param  id     Object identifier, OID.
       @return        True if successful, otherwise false.
*/
DeleteGridObject(OID oid):
       Call DeleteGridObject(oid) on PN

/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref            Grid object reference.
       @param  timeOut        Timeout value.
       @return                True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut):
       Within timeOut do
              Call Acquire(ref, timeOut, self) on PN

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref            Grid object reference.
       @param  timeOut        Timeout value.
       @return                True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut):
       Within timeOut do
              Call AcquireExclusive(ref, timeOut, self) on PN

/** Releases a previously acquired grid object referred by a grid object reference.
       @param  ref            Grid object reference, OID.
       @return                True if successful, otherwise false.
*/
Release(GridObjectRef ref):
       Call Release(ref, self) on PN

/** Receives a "CreateCopy" message and creates a copy of a GridObject on the invoked node.
       @param  o      Grid object that shall be copied on the node.
*/
Receive_CreateCopy(GridObject o):
       // creates a copy of the object instance o in the memory space of the node
       Create a clone of o
```

```
// For every primary node PNi

/** Creates a copy of the GridObject o in the Grid Universe and returns a GridObjectRef
       @param  o              The GridObject to be created on the Grid Universe.
       @param  callerNode     The caller node from where the operations has been triggered.
       @return                Reference to the created grid object. If operation fails null is
returned.
*/
CreateGridObject(GridObject o, Node callerNode):
       // Try to create on the caller node if possible
       Lookup table entry (NodeTable) where NID = callerNode
       if (entry.Load + o.Weight < entry.Capacity) then
              // create the object on the callerNode
              Call CreateCopy(o) on callerNode
              Call PN_RegisterObject(o, callerNode) to callerNode.PN
              // update distributed sequencer registration table
              PN = f(o.OID)
              Call PN_RegisterObjectStatus(o, callerNode) to PN
              return GridObjectRef(o.OID, o.GID, callerNode)
       else
              // Find a node within the universe with enough left capacity
              for each n in NodeTable
                     if (n.Load + o.Weight < n.Capacity) then
                            // create the object on the callerNode
                            Call CreateCopy(o) on n
                            Call PN_RegisterObject(o, n) on n.PN

                            // update sequencer registration table
                            PN = f(o.OID)
                            Call PN_RegisterObject(o, callerNode) on PN

                            return GridObjectRef(o.OID, o.GID, n)
                     end if
              end for
       end if
       // no capacity left in this universe
       // shared data object must be created in other universes
       for each u in UniverseTable
              if u != self then
                     GridObjectRef = Call CreateGridObject(o, u.address) on u.address
                     if (GridObjectRef != null) then
                            return GridObjectRef
                     end if
              end if
       end for

       // no free space in the entire grid universe
       return null

/** Register a grid object on a given node within a universe.
       @param  o              The GridObject to be registered on the Grid Universe.
       @param  targetNode     The node where the object has been created.
*/
PN_RegisterObject(GridObject o, Node targetNode):
       // update capacity left
       Update table entry (NodeTable) where NID = targetNode with Load = Load + o.Weight

/** Register a grid object on a given primary node where the mutual exclusion is arbitrated.
       @param  o              The GridObject to be registered on the Grid Universe.
       @param  targetNode     The node where the object has to be arbitrated.
*/
PN_RegisterObjectStatus(GridObject o, Node targetNode):
       // add object in object table
       Add table entry (NodeObjectTable, {targetNode.NID, o.GID, o.OID, READY, FALSE})
```

```
/** Finds a GridObject in the Grid Universe that has a given user provided object identifier.
        @param  id              Object identifier, OID.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 Reference to the found grid object. Returns null if no object could
be found.
*/
FindGridObject(OID id, Node callerNode):
        // Locality strategy : try to locate a node within the same universe
        Lookup table entry (NodeTable) where OID = id
        if (entry != null) then
                return GridObjectRef(entry.OID, entry.GID, entry.NID);
        end if
        // shared data object is not part of the current universe
        for each u in UniverseTable
                if u != self then
                        newref = Call FindGridObject(id, u) to u.address
                        if (newref != null) then
                                return newref
                        end if
                end if
        end for

/** Deletes all GridObjects from the Grid Universe that have a given user provided object
identifier.
        @param  id      Object identifier, OID.
        @return         True if successful, otherwise false.
*/
DeleteGridObject(OID id):
        Lookup table entry (NodeObjectTable) where OID = id
        if (entry != null) then
                Wait until entry.Status = READY
                Delete table entry (NodeObjectTable) for OID = id
                // update available capacity as object is removed
                Update table entry (NodeTable)
        end if
        // check other replicas in other universes
        for each u in UniverseTable
                if u != self then
                        Send DeleteGridObject(id) to u.address
                end if
        end for
        // wait for all replies from all universes
        Wait for all Msg
/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut, Node callerNode):
        PN = f(ref.OID)
        Call PN_Acquire(ref, timeOut, callerNode) on PN

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut, Node callerNode):
        PN = f(ref.OID)
        Call PN_AcquireExclusive(ref, timeOut, callerNode) on PN
```

```
/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
PN_Acquire(GridObjectRef ref, long timeOut, Node callerNode):
        Within timeOut do
                Lookup table entry (NodeObjectTable) where OID = ref.OID
                Wait until all entry.Status != ACQUIRE_EXCLUSIVE
                // entry state could be either READY or ACQUIRE
                // mark new acquired object as dirty
                Lookup table entry (NodeObjectTable) where GID = ref.GID
                if (entry.Dirty = TRUE) then
                        Lookup table e (NodeObjectTable) where OID = ref.OID and DIRTY = FALSE
                        // lazy object synchronization
                        Copy e.GID content to ref content
                end if
                entry.DIRTY = FALSE
                Save all entry.Status
                // all objects and their replicas have been acquired
                Set all entry.Status = ACQUIRE
        end
        if Timeout occured
                Revert Status to original state
        end if

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
PN_AcquireExclusive(GridObjectRef ref, long timeOut, Node callerNode):
        Within timeOut do
                Lookup table entry (NodeObjectTable) where OID = ref.OID
                Wait until all entry.Status = READY
                // mark new acquired object as dirty
                Lookup table entry (NodeObjectTable) where GID = ref.GID
                if (entry.Dirty = TRUE) then
                        Lookup table e (NodeObjectTable) where OID = ref.OID and DIRTY = FALSE
                        // lazy object synchronization
                        Copy e.GID content to ref content
                end if
                entry.DIRTY = FALSE
                // mark as invalid all other entries except this one
                Lookup table entry (NodeObjectTable) where OID = ref.OID and GID != ref.GID
                Set all entry.DIRTY = TRUE
                // all objects and their replicas have been acquired exclusively
                Set all entry.Status = ACQUIRE_EXLUSIVE
        end
        if Timeout occured
                Revert Status
        end if

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference, OID.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref, Node callerNode):
        PN = f(ref.OID)
        Call PN_Release(ref, callerNode) on PN

PN_Release(GridObjectRef ref, Node callerNode):
        Lookup table entry (NodeObjectTable) where OID = ref.OID
        // all objects and their replicas refered by ref have been released
        Set all entry.Status = READY
```

A.3 Centralized/Naimi-Trehel Multi-Token Algorithm

```
// For every node Ni which has following local variables:
Node  : self          - keeps the node identification (Ni)
Node  : PN            - keeps the primary node identity (e.g. address)
Long  : capacity      - free node capacity in bytes
Boolean : exclusive   - specifies if the previous acquire request was exlusive or not.
RWMutex : rwMutex[]   - read/write mutex for local arbitration according to entry-consistency
                        semantics, one per each object type

/** Creates a copy of the GridObject o in the Grid Universe and returns a GridObjectRef
       @param  o      The GridObject to be created on the Grid Universe.
       @return        Reference to the created grid object. If operation fails null is returned.
*/
CreateGridObject(GridObject o):
       Call CreateObject(o, self) on PN

/** Finds a GridObject in the Grid Universe that has a given user provided object identifier.
       @param  oid    Object identifier, OID.
       @return        Reference to the found grid object. Returns null if no object could be
found.
*/
FindGridObject(OID oid):
       Call FindGridObject(oid, self) on PN

/** Deletes all GridObjects from the Grid Universe that have a given user provided object
identifier.
       @param  oid    Object identifier, OID.
       @return        True if successful, otherwise false.
*/
DeleteGridObject(OID oid):
       Call DeleteGridObject(oid) on PN

/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
       @param  ref           Grid object reference.
       @param  timeOut       Timeout value.
       @return               True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut):
       if (Type(ref) == READONLY)
              return true
       else if (Type(ref) == PRIVATE)
              return rwMutex[ref.OID].ReadLock()
       else if (Type(ref) == MIGRATORY)
              GridObjectRef newRef = LookupRef(ref)
              if (ref.DataNode != newRef.DataNode)
                     // Object has been migrated, forward request to new node
                     return Call Acquire(newRef, timeOut) on ref.DataNode
              end if
              rwMutex[ref.OID].ReadLock()
       end if

       exclusive = false
       Within timeOut
              Send Acquire(ref, timeOut, self) on PN
              // wait until Acquired message is received
              Wait for Acquired
              return true
       end
       // timeout occured
       return false
```

```
/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut):
        if (Type(ref) == READONLY)
                return true
        else if (Type(ref) == PRIVATE)
                return rwMutex[ref.OID].WriteLock()
        else if (Type(ref) == MIGRATORY)
                GridObjectRef newRef = LookupRef(ref)
                if (ref.DataNode != newRef.DataNode)
                        // Object has been migrated, forward request to new node
                        return Call Acquire(newRef, timeOut) on ref.DataNode
                end if
                // Object resides on this node
                rwMutex[ref.OID].WriteLock()
                // The requesting node must be different to the node where the object resides
                if (ref.ClientNode != self)
                        // identify the referred grid object
                        GridObject o = ref.GetObject()
                        // Migrate the referred object from this node to the requestor node
                        ok = Call CreateGridObject(o, ref.ClientNode) on ref.ClientNode.PN
                        if (ok = true)
                                // unregister the object from its current location
                                Call PN_RemoveGridObject(ref.OID) on PN
                                capacity = capacity – size(o)
                                delete o
                                // update reference's data node as the new node
                                // remote assignment to the process space where the ref belongs to
                                ref.DataNode = ref.ClientNode
                                // obtain the mutex on the new node where the object was migrated
                                // all references will be pointed to the new node from now on
                                Call AcquireExclusive(ref, timeOut) on ref.ClientNode
                                // operations on this node are complete as the object is migrated
                                rwMutex[ref.OID].ReleaseLock()
                        end if
                end if
        end if

        exclusive = true
        Within timeOut
                Send AcquireExclusive(ref, timeOut, self) on PN
                // wait until AcquiredExclusive message is received
                Wait for AcquiredExclusive
                return true
        end
        // timeout occured
        return false

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference, OID.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref):
        if (Type(ref) == READONLY)
                return true
        else if (Type(ref) == PRIVATE)
                return rwMutex[ref.OID].ReleaseLock()
        else if (Type(ref) == MIGRATORY)
                return rwMutex[ref.OID].ReleaseLock()
        end if

        Send Release(ref, self, exclusive) on PN
        // wait until Released message is received
        Wait for Released
```

```
/** Creates a copy of a GridObject in the storage memory of the node.
        @param  o       Grid object that shall be copied on the node.
*/
CreateCopy(GridObject o):
        // creates a copy of the object instance o in the memory space of the node
        Create a clone of o
        capacity = capacity + size(o)

/** Looks up the correct node where the data referred by the reference has been migrated.
        @param  ref             Grid object reference.
        @return                 The correct reference.
*/
LookupRef(GridObjectRef ref)
        // identify the referred grid object
        GridObject o = ref.GetObject()
        if (o = null)
                // object has been migrated
                return Call FindGridObject(ref.OID, ref.ClientNode) on ref.ClientNode.PN
        end if
        return ref

// For every primary node PNi which has following local variables:
Node    : self          - keeps the node's identification (PNi)
Node    : electedNode   - elected initial primary node to hold all tokens
Boolean : requesting[]  - specifies if token for an OID has been requested or not
Node    : next[]        - array of next nodes to receive the token for an OID

Struct{
        Boolean : exclusive     - specifies if the token is owned exclusively
        Node    :  nonex[]      - nodes that hold the token in non-exclusive mode
        Node    :  latest[]     - nodes that hold the most recent copy of an object with OID
        Node    :  consumers[] – reader nodes for Producer-Consumer and Read-Mostly types

}Token : token[]        - array of tokens for each object identifier

PrimaryNode     : owner[]       - specifies the primary node that owns the token[OID]
Timer           : prTimer[]     - timer for token preeemption for each token[OID]
int             : PREEMPT_TIME – preemption time for token exchange

/** Creates a copy of the GridObject in the Grid Universe and returns a GridObjectRef
        @param  o               The GridObject to be created on the Grid Universe.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 Reference to the created grid object. Returns null in case of
error.
*/
CreateGridObject(GridObject o, Node callerNode):
        // Try to create on the caller node if possible
        Lookup table entry (NodeTable) where NID = callerNode
        if (entry.Load + o.Weight < entry.Capacity) then
                // create the object on the callerNode
                Call CreateCopy(o) on callerNode
                Call PN_RegisterObject(o, callerNode) on callerNode.PN
                // initialize the token
                for each u in UniverseTable
                        Call PN_Initialize(o.OID, callerNode) on u
                end for
                return GridObjectRef(o.OID, o.GID, callerNode, callerNode)
        else
                // Find a node within the universe with enough left capacity
                for each n in NodeTable
                        if (n.Load + o.Weight < n.Capacity) then
                                // create the object on the callerNode
                                Call CreateCopy(o) on n
                                Call PN_RegisterObject(o, n) on n.PN
                                // initialize the token
                                for each u in UniverseTable
                                        Call PN_Initialize(o.OID, n) on u
                                end for
                                return GridObjectRef(o.OID, o.GID, n, callerNode)
                        end if
```

```
                end for
        end if
        // no capacity left in this universe
        // shared data object must be created in other universes
        for each u in UniverseTable
                if u != self then
                        newref = Call CreateGridObject(o, u.address) on u.address
                        if (newref != null) then
                                return newref
                        end if
                end if
        end for
        // no free space in the entire grid universe
        return null

/** Register a grid object on a given node within a universe.
        @param  o               The GridObject to be registered on the Grid Universe.
        @param  targetNode      The node where the object has been created.
*/
PN_RegisterObject(GridObject o, Node targetNode):
        Lookup table entry (NodeTable) where NID = targetNode
        if (entry != null) then
                // update capacity left
                Update table entry (NodeTable) where NID = targetNode with Load = Load + o.Weight
        else
                // add a new entry
                Add table entry (NodeObject, {targetNode, self, self.capacity, o.Weight})
        end if
        // add object in object table
        Add table entry (NodeObjectTable, {targetNode.NID, o.GID, o.OID})

/** Finds a GridObject in the Grid Universe that has a given user provided object identifier.
        @param  id              Object identifier, OID.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 Reference to the found grid object. Returns null if no object could
be found.
*/
FindGridObject(OID oid, Node callerNode):
        // Locality strategy : try to locate a node within the same universe
        Lookup table entry (NodeTable) where OID = oid
        if (entry != null) then
                return GridObjectRef(entry.OID, entry.GID, entry.NID, callerNode);
        end if

        // shared data object is not part of the current universe
        for each u in UniverseTable
                if u != self then
                        newref = Call FindGridObject(id, u) to u.address
                        if (newref != null) then
                                return newref
                        end if
                end if
        end for

/** Deletes all GridObjects from the Grid Universe that have a given user provided object
identifier.
        @param  oid     Object identifier, OID.
        @return         True if successful, otherwise false.
*/
DeleteGridObject(OID oid):
        if owner[oid] != 0 then
                // the node does not have the token
                Send Request_AcquireExclusive(ref, callerNode) to owner[oid]
                Wait for Token(oid)
        else
                // the node has the token
                Wait until token[oid].nonex is empty
                // wait until the object is released
                Wait until token[oid].exclusive = false
        end if
```

```
        // we have the token and no clients are neither writing nor reading
        for all u in UniverseTable
                Call PN_RemoveGridObject(id) on u.address

/** Removes an object from the universe information table.
        @param  oid     Object identifier, OID.
*/
PN_RemoveGridObject(OID oid):
        Lookup table entry (NodeObjectTable) where OID = oid
        if (entry != null) then
                Delete table entry (NodeObjectTable) for OID = oid
                // update available capacity as object is removed
                Update table entry (NodeTable)
        end if

/** Acquires non-exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref                     Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut                 Timeout value.
        @return                         True if successful, otherwise false.
*/
Acquire(GridObjectRef ref, long timeOut, Node callerNode):
        queue = queue + (OP_Acquire, ref, timeOut, callerNode)

/** Acquires exclusively within the specified timeout[ms] the grid object referred by a grid
object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @param  timeOut         Timeout value.
        @return                 True if successful, otherwise false.
*/
AcquireExclusive(GridObjectRef ref, long timeOut, Node callerNode):
        queue = queue + (OP_AcquireExclusive, ref, timeOut, callerNode)

/** Processes continuously requests from the request queue.
*/
PN_ProcessQueue():
        while (true) do
                entry = queue.head
                queue = queue - head
                if (entry == OP_Acquire)
                        PN_Acquire(entry.ref, entry.callerNode)
                        if (queue not empty) and (queue.head.entry == OP_Acquire)
                                // eager synchronization if next operation is acquire
                                PN_EagerSync(queue.head.ref, queue.head.callerNode)
                        end if
                else if (entry = OP_AcquireExclusive)
                        PN_AcquireExclusive(entry.ref, entry.callerNode)
        end while

/** Initializes the primary node data stuctures for a token type.
        @param  oid     Token identifier to be initialized.
        @param  n       The node where the an object of type oid was created.
*/
PN_Initialize(OID oid, Node n):
        requesting[oid] = false
        next[oid] = 0
        token[oid].exclusive = false
        if (self = electedNode) then
                owner[oid] = 0                  // this node is the owner
        else
                owner[oid] = electedNode        // electedNode is the owner
        end if
        // mark this node as one that contains the latest shared object
        Add n to token[oid].latest
```

```
/** Performs an acquire request.
        @param  ref           Grid object reference.
        @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_Acquire(GridObjectRef ref, Node callerNode):
        if (Type(ref) == RES)
                requesting[ref.OID] = true
                if owner[ref.OID] != 0 then
                        // the node does not have the token
                        // token must be requested
                        Send Request_AcquireExclusive(ref, callerNode) to owner[ref.OID]
                        owner[ref.OID] = 0
                        Wait for Token(ref.OID, latestNodes)
                else
                        // the node has the token
                        Wait until (token[ref.OID].nonex is empty)
                        Wait until (token[ref.OID].exclusive == false)
                end if
                // perform object state synchronization
                Call AssembleObject(ref, token[ref.OID].nonex) on ref.DataNode
                // reset token
                token[ref.OID].latest = ref.DataNode
                token[ref.OID].nonex = 0
                token[ref.OID].exclusive = true
                Send Acquired to callerNode
        end if

        // regular handling
        if owner[ref.OID] != 0 then
                // the node does not have the token
                // node registers to the token owner as non-exclusive access
                Send Request_Acquire(ref, callerNode, self) to owner[ref.OID]
                Wait for Acquire_Granted(ref.OID, latestNodes)
        else
                // this primary node has the token
                Wait until (token[ref.OID].exclusive == false)
                // now the token can be held non-exclusively
                Add callerNode to token[ref.OID].nonex

                latestNodes = token[ref.OID].latest
                if (ref.DataNode not in latestNodes) then
                        Add ref.DataNode to token[ref.OID].latest
                end if

                // update consumer nodes if not already known
                if ((Type(ref) == PC or RM) and callerNode not in token[ref.OID].consumers)
                        Add callerNode to token[ref.OID].consumers
                end if
        end if
        // synchronize object if not in latest
        if (ref.DataNode not in latestNodes)
                // update the object pointed by ref residing on callerNode
                // with one value of objects residing on latestNodes
                Synchronize(ref, ref.DataNode, latestNodes)
        end if
        Send Acquired to callerNode

/** Performs an acquire exclusive request.
        @param  ref           Grid object reference.
        @param  callerNode    The caller node from where the operations has been triggered.
*/
PN_AcquireExclusive(GridObjectRef ref, Node callerNode):
        if (Type(ref) == RES)
                requesting[ref.OID] = true
                if owner[ref.OID] != 0 then
                        // the node does not have the token
                        // node registers to the token owner as non-exclusive access
                        Send Request_Acquire(ref, callerNode) to owner[ref.OID]
                        Wait for Acquire_Granted(ref.OID, latestNodes)
                else
```

```
                        // this primary node has the token
                        Wait until (token[ref.OID].exclusive == false)

                        // now the token can be held non-exclusively
                        Add callerNode to token[ref.OID].nonex
                end if
                Send AcquiredExclusive to callerNode
                return
        end if

        // regular handling
        requesting[ref.OID] = true
        if owner[ref.OID] != 0 then
                // the node does not have the token
                // token must be requested
                Send Request_AcquireExclusive(ref, callerNode, self) to owner[ref.OID]
                owner[ref.OID] = 0
                Wait for Token(ref.OID, latestNodes)
        else
                // this primary node has the token
                Wait until (token[ref.OID].nonex is empty)
                Wait until (token[ref.OID].exclusive == false)
                latestNodes = token[ref.OID].latest
        end if

        // update read-mostly nodes if not already known
        if (Type(ref) == RM and callerNode not in token[ref.OID].consumers)
                Add callerNode to token[ref.OID].consumers
        end if

        // synchronize objects if not in latest
        if (ref.DataNode not in latestNodes)
                // update the object pointed by ref residing on callerNode
                // with one value of objects residing on latestNodes
                Synchronize(ref, ref.DataNode, latestNodes)
        end if
        token[ref.OID].latest = ref.DataNode
        token[ref.OID].nonex = 0
        token[ref.OID].exclusive = true
        Send AcquiredExclusive to callerNode

/** Processes an acquire request.
        @param  ref           Grid object reference.
        @param  requestorNode  The caller node from where the operations has been requested from.
        @param  callerPN      Primary node from where the request has been issued.
*/
Receive_Request_Acquire(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if (owner[ref.OID] == 0) then
                // this node has the token
                Wait until (token[ref.OID].exclusive == false)
                Add requestorNode to token[ref.OID].nonex
                // send the token granted to the requesting primary node
                Send Acquire_Granted(ref.OID, token[ref.OID].latest) to callerPN
                Add ref.dataNode to token[ref.OID].latest
                // update consumer nodes if not already known
                if ((Type(ref) == PC or RM) and requestorNode not in token[ref.OID].consumers)
                        Add requestorNode to token[ref.OID].consumers
                end if
        else
                // non-root node, forward request
                Send Request_Acquire(ref, requestorNode) to owner[ref.OID]
        end if

/** Processes an acquire exclusive request.
        @param  ref           Grid object reference.
        @param  requestorNode  The caller node from where the operations has been requested from.
        @param  callerPN      Primary node from where the request has been issued.
*/
Receive_Request_AcquireExclusive(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if owner[ref.OID] = 0 then
```

```
                // serve internal requests with priority within a certain time frame
                if (queue has requests with ref.OID) then
                        // If timer is not already setup, it must be started
                        if (prTimer[ref.OID] is not running) then
                                prTimer[ref.OID].start(PREEMPT_TIME)
                                return
                        end if

                        // Check if the timer expired
                        if (prTimer[ref.OID] is not expired) then
                                return
                        end if
                        // the timer expired, thus we have to process requests
                end if

                // update read-mostly nodes if not already known
                if (Type(ref) == RM and requestorNode not in token[ref.OID].consumers)
                        Add requestorNode to token[ref.OID].consumers
                end if

                // time frame elapsed or no pending internal requests
                if (requesting[ref.OID] == true) then
                        // The node asked for acquire exclusive
                        next[ref.OID] = callerPN
                else
                        Wait until (token[ref.OID].nonex is empty)
                        Wait until (token[ref.OID].exclusive == false)
                        // send the token directly to the requesting node
                        Send Token(ref, token[ref.OID].latest) to callerPN
                        // reset token
                        token[ref.OID].exclusive = false
                        token[ref.OID].latest = empty
                        token[ref.OID].nonex = empty
                end if
        else
                // non-root node, forward request
                Send Request_AcquireExclusive(ref, requestorNode, callerPN) to owner[ref.OID]
        end if
        owner[ref.OID] = callerPN

/** Releases a previously acquired grid object referred by a grid object reference.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been triggered.
        @return                 True if successful, otherwise false.
*/
Release(GridObjectRef ref, Node callerNode):
        requesting[ref.OID] = false
        if (token[ref.OID].exclusive == true) then
                token[ref.OID].latest = callerNode
                // synchronize all “consumer nodes”
                if (Type(ref) == PC or RM)
                        Call SynchronizeAll(ref, token[ref.OID].consumers) on ref.DataNode
                end if
                // release from exclusive acquire
                if (next[ref.OID] != 0) then
                        // Send the token if the timer expired
                        if (prTimer[ref.OID] is expired) then
                                owner[ref.OID] = next[ref.OID]
                                Send (Token, ref, token[ref.OID].latest) to next[ref.OID]
                                next[ref.OID] = 0
                                // reset token
                                token[ref.OID].latest = empty
                                token[ref.OID].nonex = empty
                        end if
                end if
                token[ref.OID].exclusive = false
        else
                // release from non-exclusive acquire
                if (owner[ref.OID] != self)
                        // primary node does not have the token
```

```
                        Send (Request_Release, callerNode, false) to owner[ref.OID]
                        Wait for message (Release_Granted)
                else
                        Remove callerNode from token[ref.OID].nonex
                end if
        end if
        Send Released to callerNode

/** Processes a release request.
        @param  ref             Grid object reference.
        @param  requestorNode   The caller node from where the operations has been requested from.
        @param  callerPN        Primary node from where the release is issued from.
 */
Receive_Request_Release(GridObjectRef ref, Node requestorNode, PrimaryNode callerPN):
        if (owner[ref.OID] == 0) then
                // this node has the token
                if (exclusive == false) then
                        Remove requestorNode from token[ref.OID].nonex
                end if
                // send the token granted to the requesting primary node
                Send Release_Granted[ref.OID] to callerPN
        else
                // non-root node, forward request
                Send Request_Release(ref, requestorNode, callerPN) to owner[ref.OID]
        end if

/** Processes a token received message.
        @param  ref             Grid object reference.
        @param  nodes           Array of nodes that have the latest object values.
*/
Receive_Token(GridObjectRef ref, Node[] nodes):
        // receive the token from node
        owner[ref.OID] = 0
        Add nodes to token[ref.OID].latest

/** Synchronizes an object data with one of the latest object values.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been requested from.
*/
PN_EagerSync(GridObjectRef ref, Node callerNode):
        if owner[ref.OID] != 0 then
                // the primary node does not have the token
                // node registers to the token owner as non-exclusive access
                Send Request_LatestNodes(ref, callerNode) to owner[ref.OID]
                Wait for LatestNodes(ref.OID, latestNodes)
        else
                // this primary node has the token
                latestNodes = token[ref.OID].latest
        end if

        if (callerNode not in latestNodes)
                // update the object pointed by ref residing on callerNode
                // with one value of objects residing on latestNodes
                Synchronize(ref, callerNode, latestNodes)
        end if

/** Processes an acquire request.
        @param  ref             Grid object reference.
        @param  callerNode      The caller node from where the operations has been requested from.
*/
Receive_LatestNodes(GridObjectRef ref, Node callerNode):
        // callerNode is the requesting node
        if (owner[ref.OID] == 0) then
                // this primary node has the token
                // send the token's latest node list granted to the requesting node
                Send LatestNodes(ref.OID, token[ref.OID].latest) to callerNode
                Add ref.dataNode to token[ref.OID].latest
        else
                // non-root node, forward request
                Send Request_LatestNodes(ref, callerNode) to owner[ref.OID]
```

```
        end if


/** Assembles the object state residing on this node out of the updated values of replicas
residing in a list of nodes.
        @param  ref           Grid object reference.
        @param  nodes         Array of nodes that hold updated versions of object parts.
*/
AssembleObject(GridObjectRef ref, Node[] nodes):
        // Construct the parts of the object
        GridObject o = ref.GetObject()
        Part parts[] = o.GetParts()
        Part changedPart

        for each part in parts
                index++ = 0
                for each n in nodes
                        // part is the local part that might have been written somewhere
                        // changedPart is the out parameter, the most recent written part
                        changed = (Call GetObjectPart(index, part, changedPart) on n //remote call
                        if (changed = true)
                                parts[index] = changedPart
                                // first found is the correct non-conflicting part
                                break
                        end if
                end for
        end for
```