Experiences on Grid Shared Data Programming

Dacian Tudor*, Georgiana Macariu, Wolfgang Schreiner, Vladimir Cretu**

"Politehnica" University of Timisoara, Computer Science and Engineering Department, Vasile Parvan Street, No. 2, 300223, Timisoara, Romania
E-mail: dacian@cs.upt.ro, georgiana@cs.upt.ro, vcretu@cs.upt.ro

*Corresponding author

**
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, 4040 Linz, Austria
E-mail: Wolfgang.Schreiner@risc.uni-linz.ac.at

Abstract: Despite the continuous advances of the last years in grid computing, programming paradigms are dominated by the message passing concept. There is little support for other paradigms such as shared data or associative programming. In this paper we analyze why previous attempts did not have a significant impact in the grid computing community. We start by assessing the landscape of grid programming solutions with a focus on shared data concepts. Next, we introduce an original idea to attack shared data programming on the grid by making use of both relaxed consistency models and user specified type consistency in an object oriented model. Last but not least, we present a prototype architecture together with experimental results.

Keywords: shared memory programming; grid programming; distributed shared memory.

Biographical notes: Dacian Tudor holds a MsC in Computer Science at the "Politehnica" University of Timisoara, Computer Science Department where he is working as a research group leader. Currently he focuses on creating a grid service layer for shared data programming tailored for wide scale distributed systems. His research topics include distributed and grid computing as well as mobile communication systems.

Georgiana Macariu is a PhD student at the "Politehnica" University of Timisoara. Her research interests are grid and distributed computing as well as mobile communication systems.

Prof. Dr. Wolfgang Schreiner is Associate Professor at the Research Institute for Symbolic Computation (RISC) in Linz, Austria. His research interests are in parallel and distributed computing, formal methods in computer science, and e-learning.

Prof. Dr. Eng. Vladimir-Ioan Cretu is full professor and head of the Department of Computer Science and Engineering, Faculty of Automation and Computers. His research interests cover real-time and distributed systems, software for data acquisition and processing systems for electrical machines instrumentation and measurement, data structures, algorithm design and analysis, embedded systems, and software development processes and technologies.

1 INTRODUCTION

Most of the currently available grid programming models rely on foreign programming models that are adopted via a technology porting process. Here we include MPI-like libraries like MPICH-G2 (Karonis, Toonen, and Foster, 2002), file based distributed data services based on GridFTP (Allcock et. al., 2001) and higher level data access services like OGSA-DAI (Karasavvas et al., 2005). 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 new 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 to a certain number of nodes and work best with 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 models 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.

In this paper we aim to identify the challenges of grid shared data system design and the reasons why these kinds of systems did not have a significant impact on the grid programming community. We start by elaborating in Section 2 the assessment criteria for general grids as well as grid shared data systems. In Section 3, we highlight the most important grid shared data attempts and emphasize their weaknesses. In Section 4, we present our design strategy in order to achieve a widely distributed, scalable and efficient grid shared data system that provides a generic and flexible object oriented programming interface.

2 ASSESSMENT CRITERIA

Based on literature study of a different grid shared data based systems, we tried to abstract and identify some of their properties which are directly related to the programming model they introduce, their adaptation level towards the grid specific interfaces and their performance behavior in various conditions. The first five criteria are based on (Lee et al., 2001) and they reflect system specific criteria. The following five criteria are abstracted out of our literature based observations we made on several grid systems and they reflect aspects which are most of the time overlooked during the system design or during the system assessment and performance measurements.

2.1 Usability and transparency

Grid shared data abstractions must be suitable for various types of problem domains from local computing to large scale high performance computing. There should be no constraints in building codes that are targeted to a specific architecture so that different development paths are followed depending on the system's requirements and architecture. Next, access to the shared data shall be done through a generic and transparent interface, which shall not require tailoring for different usage scenarios.

2.2 Dynamic and heterogeneous configurations

Dynamic and heterogeneous environments that change frequently due to machine availability, new connection paths, different communication latencies due to connection changes, new available resources are common assumptions for any grid system. As a consequence, predefined built-in logic is not suitable for large scale grid systems as it obstructs reconfigurations and system evolution.

2.3 Portability and interoperability

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. Interoperable grid systems are based on open standards and protocols. Ideally, the protocols, services and interfaces that realize the grid shared data model shall expose interoperable concepts as well.

2.4 Reliability and fault tolerance

Reliability and fault tolerance are general system desiderates for any grid system with a certain degree of determinism. Most of the time reliability is associated with performance reliability, meaning that multiple code executions shall not have significant performance deviations. Addressing these issues in the application layer is not applicable anymore as grids aim to expose high level functionality with advanced management support. Ideally, these characteristics shall be part of the run-time mechanisms of the grid system.

2.5 Security and privacy

As grids span between virtual organizations with 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 system and less visible on the programming interfaces.

2.6 Flexible replication techniques

Replication techniques have been used in case of service-centric system to increase service availability through resource redundancy. Data-centric systems like shared data systems make use of replication for performance. Depending on the replication techniques and replica synchronization protocols, it is expected to observe significant performance differences. We believe that a key point in assuring reliable performance is to adapt replication decisions and algorithms to specific use cases. More, we do not see the replication decisions as part of the system, but more like information collection from the application side based on predefined system metrics.

2.7 Replica consistency and coherence

It is generally accepted that strong consistency specifications like sequential consistency (Lamport, 1979) are not suitable to large scale systems such as a grid. Relaxed consistency specifications like entry consistency (Bershad, Zekauskas and Sawdon, 1993) have a much higher chance to perform better on grid systems. However, we believe that consistency alone does not provide sufficient information, as it basically bounds the system to a replica update scheme. Having flexible replication schemes would be a clear advantage in accommodating different shared data usage scenarios.

2.8 Mutual exclusion

Grid shared data systems introduce the problem of data sharing and mutual exclusion. Mutual exclusion algorithms have been developed during the last decades, but there is little information on their performance behavior when applied to the grid level. Some recent investigations conducted in (Sopena et al., 2007) and (Bertier, Arantes and Sens, 2004) provide an empirical performance evaluation of an extended version of the Naimi-Trehel (Naimi, Trehel, and Arnold, 1996) and two level compositional algorithms that realize mutual exclusion on the grid level. We believe that the performance of grid shared data solution is directly determined by the choice of the mutual exclusion algorithm on the grid level.

2.9 Wide scale and extreme conditions

We believe that grids consisting on several strongly connected clusters via fast interconnections do not provide a suitable environment for performance analysis. Instead we focus on really wide scale systems, which are dominated by large latency connections. The motivation for such criteria is simply because of the network saturation effect which becomes obvious if grids become public, instead being isolated and used as dedicated resources. We believe that only on a wide scale the original definition (Foster and Kesselman, 1998) and purpose of grid systems is truly met.

2.10 System verification and performance analysis

Most of the grid shared data systems we have surveyed by literature study, such as Dedysis (Osrael, Froihofer, and Goeschka, 2006) and JUXMEM (Antoniu, Bougé, and Mathieu, 2005), have been evaluated empirically from the performance point of view. A critique perspective on these systems is presented in Section 3. In case of the analyzed systems, either some consecrated parallel algorithms were run on several configurations, or an artificially created problem that used the system was deployed in order to collect the performance results. Most of the times, the conditions during the experiment were not described, suggesting that clusters were used as dedicated resources. We believe that only system verification and performance analysis through analytical and formal methods can reveal more meaningful aspects about the system behaviour. In addition, real situation experiments are quite time consuming to run in various configurations. As some experiments might require hours to run, running a complete experiment suite can easily span to days or weeks. Opposite to this situation, system verification and performance analysis via computer-aided verification tools can dramatically reduce the simulation time and enable simulation scenarios that are impossible to provide in real-life situations.

3 EXISTING GRID SOLUTIONS LANDSCAPE

When it comes to programming grid applications, there are not so many choices of programming paradigms. Most of the grid based projects that we have encountered make extensive use of message passing techniques either as a grid-integrated solution like MPICH-G2 (Karonis, Toonen and Foster, 2002) or solutions that simply use the grid as an execution environment. The research landscape for shared data programming on the grid is at its dawn. We believe that some of the major obstacles in the development of this paradigm are coming from the complexity of the solution and some challenges described by some of the assessment criteria. More specifically, we believe that the combination of replication techniques, mutual exclusion and consistency replication is a major challenge in building scalable and efficient shared data grid systems. Last but not least, the lack of rigorous system analysis makes previous experiment results questionable in large scale and extreme conditions.

3.1 File and catalogue based solutions

Grid middleware solutions like Globus (Foster, and Kesselman, 1997) provide mechanisms for performing file replication and replica location services via hierarchical catalogs. Unfortunately, these mechanisms are suitable only for immutable data handling and not dynamic data as required by shared data programming paradigm. Several services for replica consistency handling have been proposed based on grid middleware, but to our knowledge, none of them is providing a programming interface and integration into the grid environment. In other words, the solutions remain particular to certain scenarios, without having the required generic level aimed by a programming paradigm.

3.2 Dedisys

Dedisys (Osrael, Froihofer, and Goeschka, 2006) is one of the very recent research activities towards replicated data systems across the grid, where the primary focus is increased availability by sacrificing consistency. The core idea is to continue the normal execution when faults occur, such as network partitions due to disconnected communication paths, and provide reconciliation points when connections are restored. In this way different replicas

could evolve independently at the reconciliation price paid by the programmer which has to provide the reconciliation logic. Based on the information we collected on Dedisys, it is not clear how the system performs data replication, but we suspect that it is bound at design time and cannot be adjusted based on the application requirements. Second, it is not clear how mutual exclusion is realized. Last but not least, system assessment seems to be done on an experimental basis, but further performance evaluations might follow.

3.3 JUXMEM

JUXMEM (Antoniu, Bougé, and Mathieu, 2005), is a recent grid sharing data solution based on a peer-to-peer middleware architecture that provides a transparent and generic interface to shared data programming on the grid. It uses the JXTA (Seigneur, Biegel, and Damsgaard, 2003) middleware to provide a sharing service for distributed shared data. JUXMEM does not provide any data structure, but rather a flat view provided as a memory buffer where the user has to map its own data representation. The memory consistency protocol is entry consistency and replication decision is fixed and bound at the time the shared data is created. This means that a high data usage does not lead to dynamic replication, but it relies on its already existing replicas. The system takes care of failure conditions by promoting grid nodes to new roles in order to fulfill the system specification.

JUXMEM's validation has been performed on experimental basis, by integration into the Grid-RPC DIET (Caron and Desprez, 2006) environment and running different parallel applications as benchmarks. From the set of run experiments, it is not clear how the system behaves on a large scale deployment and large latency connections. Based on the author's descriptions of the required adaptations in the JXTA middleware and the fact that all the constructs are based on peer-to-peer communication protocols, we feel that the peer-to-peer layer might lead to a performance limiting factor on large scale grid systems.

4 A NEW APPROACH

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 modelled 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 others within the same group with the same known and upper bound latency. Thus, we consider together all machine groups and depict them as a single group node as shown in Figure 1. Such hierarchical structure is constantly getting deeper (more levels) and wider (more groups) during the evolution of grid systems. 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 we consider that there is an upper and lower latency bound.

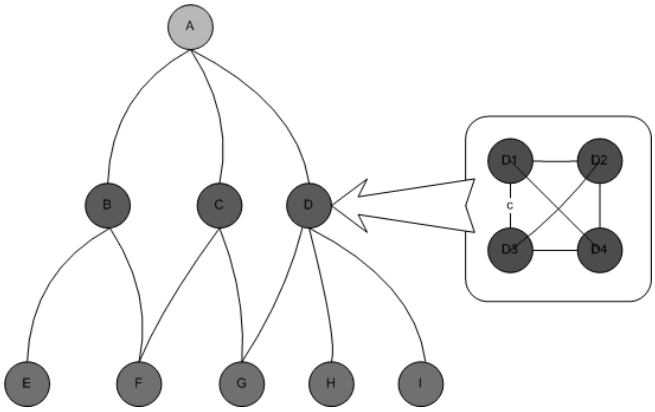

Figure 1 *Grid Node Layering*

We follow two directions to overcome the problem of communication delay from different perspectives. First, we reconstruct the problem and partition it into acceptable communication groups with known latencies. Second, we provide useful programming information so that the run-time system can take advantage of semantic 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.

4.1 System separation

Some of the previous attempts in designing DSM for the grid have used logical mappings over one single large machine group. As presented in section three, in case of the JUXMEM approach, where peer-to-peer groups have been spawned across the grid, the authors of JUXMEM recognized that the wide distribution of peers in the overlay layer is problematic and current overlay implementations such as JXTA (Seigneur, Biegel, and Damsgaard, 2003) 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 relaying on a predefined mapping.

In order to address thousands of widely distributed nodes, we decompose the system into a federation of logical groups 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. Of course, the physical entities that form a universe could be heterogeneous (e.g. machines with different resources and operating systems). Communication outside the universe, or between universes, is unbounded, but still it has an upper limit.

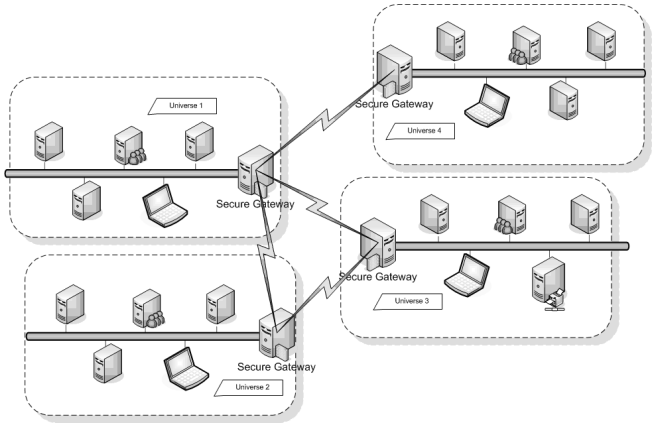

Figure 2 *Physical Universe Mapping Sample*

In our view 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 RAM 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. 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. A sample grid universe consisting of four universes mapped to four clusters is depicted in Figure 2.

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 large latencies. The system decides at run-time to replicate the data based on its specification. 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.

4.3 Consistency protocol

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.

We have evaluated several usage scenarios to such a shared data model and we consider as the most promising consistency model the entry consistency model, 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.

4.4 Type coherence

We tackle the problem of grid shared data in two distinct dimensions. First, we have considered the consistency model 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. Based on the scenarios we have selected nine object types: read-only, private, migratory, producer-consumer, read-mostly, result, write-mostly, generic and synchronization object.

The programming model we specified defines the above different types of grid shared objects and synchronization

mechanisms in order to give the run-time system useful information to allow a higher concurrency degree and to minimize wide area communication overhead.

4.5 Mutual exclusion

Some of the very recent activities towards better mutual exclusion algorithms on the grid have been elaborated in (Sopena et al., 2007) and (Bertier, Arantes and Sens, 2004), where the authors proposed a compositional approach and an extension to the Naimi-Trehel (Naimi, Trehel, and Arnold, 1996) algorithm. Both the proposed extension and the compositional approach could be applied to our universe structure, but there are various modifications could be necessary. 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 object access is possible. This means that if for example an existing algorithm is considered for realizing the mutual exclusion protocol, it has 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.

In (Sopena et al., 2007) the authors presented an algorithm composition to realize a hierarchical mutual exclusion protocol on an infrastructure similar to the one we have proposed. Different algorithms pairs are constructed for universe and between universe resource exclusion handling. They observed 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 significant performance impact, except the number of exchanged messages. Based on the measurements in (Bertier, Arantes and Sens, 2004), it seems that the Naimi-Trehel (Naimi, Trehel, and Arnold, 1996) 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).

Although the experiments were run in very particular environments that are neither widely distributed, nor having large latencies, we took these observation and proposed a new mutual exclusion algorithm adapted for entry consistency specification, where a multi-token Naimi-Trehel algorithm is used between universes and a type-parallel centralized algorithm is used inside each universe. We hope to explore the small latency communication within a universe and centralize information on a special node that would allow us to achieve fast decisions in terms of replica update and state identification.

4.6 System verification and prototyping

We have previously criticized the lack of rigorous system verification and performance modeling for existing grid shared data systems. The same is true for the mutual exclusion algorithms which have been evaluated through experiments on fast interconnected machines (Sopena et al., 2007). In case of (Bertier, Arantes and Sens, 2004), the system consisted of only three clusters each made of three machines. We argue that such experiments are not significant for the purpose of our goals, namely true widely distributed systems where large latency connections are dominant between universes. Thus, we argue that better instruments for system verification and performance assessment are necessary. In our current work, we aim to focus on probabilistic formal verification in order to validate our ideas and provide meaningful performance figures in both relaxed and extreme system conditions. While we focus on system verification, we put less importance on prototyping. We still aim to build a prototype and integrate it into the ProActive middleware (Baduel, Baude, and Caromel, 2002), which appears quite promising due to its highly asynchronous behavior and concept of futures.

5 GUN PROTOTYPE

GUN is the acronym for Grid UNiverse and represents a Java based implementation of the grid universe model defined in (Tudor, Cretu and Schreiner, 2008). 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 (Baker and Smith, 2001) or ProActive (Baduel, Baude, and Caromel, 2002) solutions do for example), the RMI model fits well to the abstract model.

GUN reflects the architecture of the abstract model and the abstract system architecture described in (Tudor, Cretu and Schreiner, 2008). 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. A sample deployment of a grid universe across physical machines is depicted in Figure 2.

The GUN prototype is divided into three layers, as illustrated in Figure 3. 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.

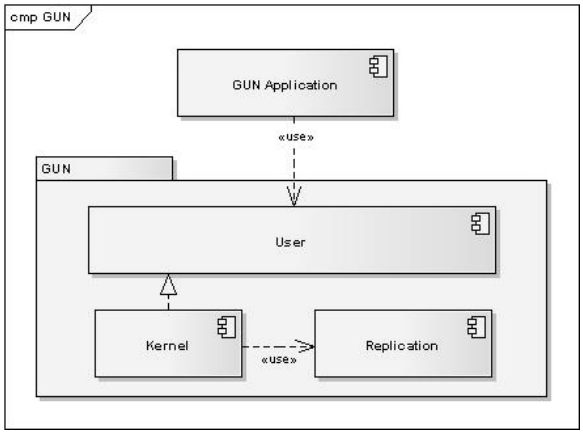

Figure 3 *GUN High Level Architecture*

The user layer depicted in Figure 4 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.

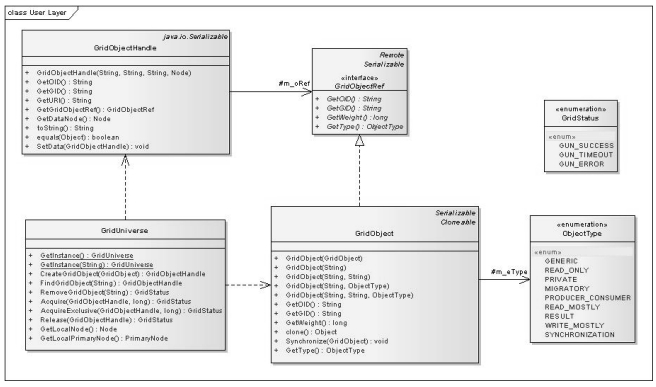

Figure 4 *GUN User Layer*

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.

When the GUN system is started, first all primary nodes are being started. Each primary node has a configuration file which contains at the address of at least other primary nodes. 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 applies 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.

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 (Tudor, Cretu and Schreiner, 2008) 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 5.

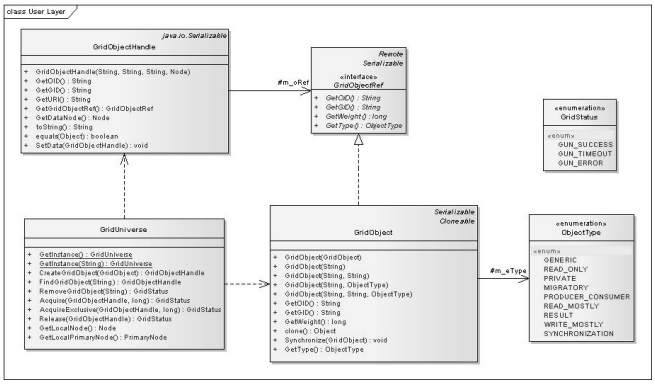

Figure 5 *GUN Kernel Messages*

When a client invokes an operation on a grid object via the GridUniverse, the node where the client resides gets, depending on the desired operation, one of the Acquire, AcquireExclusive or Release methods invoked. 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 message 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 6.

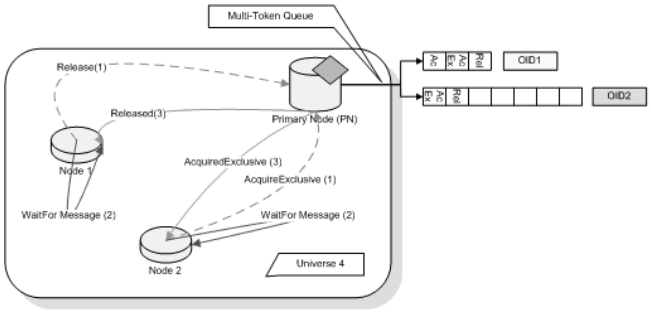

Figure 6 *GUN Message Queue*

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
- Acquire exclusive requests
- Release requests
- Token reception
- Synchronization of up-to-date nodes

The distributed mutual exclusion algorithm is based on the multi-token concept. For every group of objects 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 7. 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).

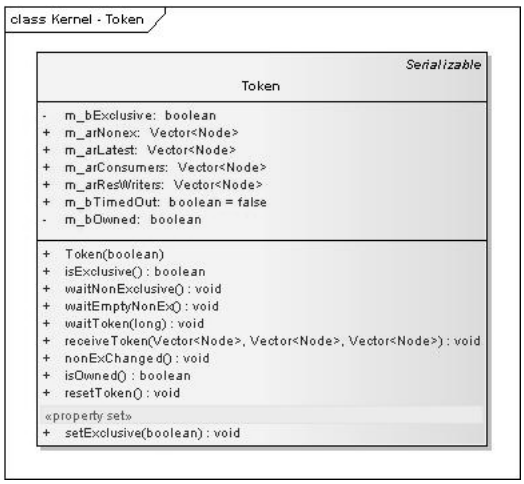

Figure 7 *GUN Token*

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.

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 8.

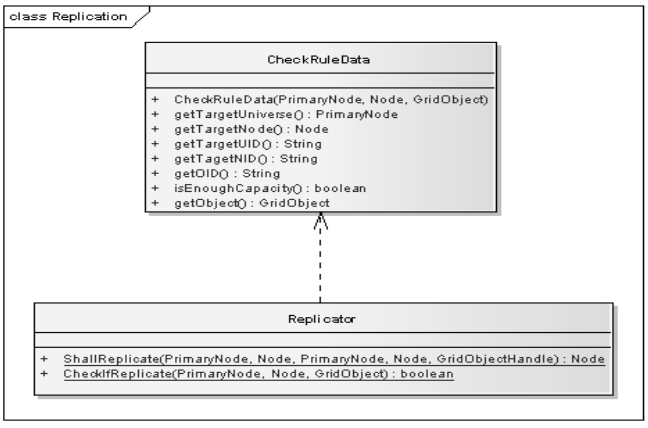

Figure 8 *GUN Replication Engine*

6 GUN REPLICATION POLICIES

Whenever a client application searches for a grid object or requests exclusive or non-exclusive access to an object, the kernel layer invokes the replication engine which may decide to replicate the object closer to the node where the client application is running. The decision of the engine is based on a set of replication policies defined at system deployment time. Each policy has an associated parameter (i.e. number of requests for exclusive or non-exclusive access to the object, number of times the object has been written by the client application, number of denied accesses) and a threshold value. The replication engine evaluates the parameter based on information provided by the monitoring layer and if its value reaches the threshold value replication of grid object is triggered. We define two types of replication policies: *universe policies* and *node policies*. For universe policies the current value of the policy parameter is determined using monitoring information collected from all nodes in the client's universe and the object replica can be created on any node in the universe, while for node policies only monitoring information about the node where the client application resides is considered and also the replica will be created on this specific node.

We define four main replication policies based on system status information. Object replication is triggered either if all policies hold true or if any of the policies holds true, as specified at system deployment.

Object Usage Degree Policy: A grid object *o* is replicated on a node or in a universe only if it was used (read or written) by that node/universe at least *UT* times. This policy assures that only objects currently used by at least one grid application are replicated and avoids replicating objects highly used in the past but not requested anymore.

Replication Policy 1: Object Usage Degree Policy
Associated Parameter: *AcquireTotal + AcquireExclusiveTotal*
Condition:
for **policyType = NODE_POLICY**
targetNode ≠ *sourceNode*
targetNode.AcquireTotal(oid) + *targetNode.AcquireExclusiveTotal(oid)* ≥ *UT*
targetNode.Count(oid) = 0
for **policyType = UNIVERSE_POLICY**
targetUniverse ≠ *sourceUniverse*
targetUniverse.AcquireTotal(oid) + *targetUniverse.AcquireExclusiveTotal(oid)* ≥ *UT*
Comments:
UT - Usage Threshold

Figure 5 *Object Usage Degree Policy*

Object Update Degree Policy: A grid object *o* is replicated on a node or in a universe only if it was written by that node/universe at least *UpT* times.

Replication Policy 2: Object Update Degree Policy
Associated Parameter: *AcquireExclusiveTotal*
Condition:
for **policyType = NODE_POLICY**
targetNode ≠ *sourceNode*
targetNode.AcquireExclusiveTotal(oid) ≥ *UpT*
targetNode.Count(oid) = 0
for **policyType = UNIVERSE_POLICY**
targetUniverse ≠ *sourceUniverse*
targetUniverse.AcquireExclusiveTotal(oid) ≥ *UpT*
Comments:
UpT - Update Threshold

Figure 6 *Object Update Policy*

Acquire Miss Degree Policy: A grid object *o* is replicated on a node or in a universe only if *MT* requests for the object issued from that node/universe could not be served within a period of interest.

Replication Policy 3: Acquire Miss Degree Policy
Associated Parameter: *AcquireMiss + AcquireExclusiveMiss*
Condition:
for **policyType = NODE_POLICY**
targetNode ≠ *sourceNode*
targetNode. AcquireMiss(oid) + *targetNode.AcquireExclusiveMiss(oid)* ≥ *MT* ∧
targetNode.Count(oid) = 0
for **policyType = UNIVERSE_POLICY**
targetUniverse ≠ *sourceUniverse*
targetUniverse.AcquireMiss(oid) + *targetUniverse.AcquireExclusiveMiss(oid)* ≥ *MT*
Comments:
MT - Acquire Miss Threshold

Figure 7 *Acquire Miss Degree Policy*

Object Count Policy: An object *o* is replicated in a universe only if the number of copies of the object in the universe does not exceed *CT*. On any node in the universe at some point in time an application may start and this application will need to create its own objects. This policy assures that the performance of the new application will not be degraded simply because there was not enough space for its objects in the universe, and thus objects were created in another universe.

Replication Policy 4: Object Count Policy
Associated Parameter: *ObjectCount*
Condition:
for **policyType = NODE_POLICY**
targetNode ≠ *sourceNode*
targetNode.Count(oid) ≤ *CT*
for **policyType = UNIVERSE_POLICY**
targetUniverse ≠ *sourceUniverse*
targetUniverse.Count(oid) ≤ *CT*
Comments:
CT - Count Threshold
UNIVERSE TYPE policies can refer to the total number of objects in the universe or to the number of objects with a given OID.
NODE TYPE policies refer only to the total number of objects on the node.

Figure 8 *Object Count Policy*

7 GUN EXPERIMENTAL RESULTS

Performance of any distributed application can be increased by improving data locality. We performed several experiments to show how the proposed policy-based replication strategy affects data locality. For our experiments we defined an application scenario whose performance is greatly influenced by this characteristic and also stresses the importance of object replication in distributed applications with aggressive concurrency. The application makes use of a shared object composed of multiple parts which are also shared objects. Basically, the application builds the object from its components using a set of worker processes distributed across a grid universe. Each object part is build by several workers in parallel and a worker builds one or several parts. While working on a part a worker requires exclusive or non-exclusive access to that part. The access mode for each part is specified using another shared object, called *BuildingRules*. The worker processes are distributed on the nodes of the universes such that workers that build common parts run in the same universe. An extended version of our experiments is presented in (Macariu, Tudor and Cretu, 2008).

For the experiments we deployed a grid universe consisting of three universes connected through a wide area network with an average of five nodes in each universe. The nodes in a universe communicate over a faster network (e.g. LAN). As the model proposed in (Tudor, Cretu and Schreiner, 2008) assumes universes are connected wide area network, we chose NistNET (Carson and Santay, 2003) for WAN emulation. The WAN emulator connects the three universes and sets a packet delay of 30 ms between the primary nodes of the universes.

Table 1 *Tested Replication Policies*

Policy P1	No replication
Policy P2	*Type: Object Count Policy* "One object per universe": a replica of the object is created in each universe where it is requested but in each universe only one copy of the object will exist (the replica is created when the object is first requested).
Policy P3	*Type: Object Usage Policy* "One object per node": a replica of the object is created on each node where it is requested (the replica is created when the object is first requested).
Policy P4	*Type: Object Update Policy and Object Count Policy* A replica of the object is created in a universe if the object has been updated in at least 20% of total requests issued from each universe (a single copy of the object might exist in each universe).
Policy P5	*Type: Object Update Policy and Object Count Policy* A replica of the object is created in a universe if the object has been updated in at least 40% of total requests issued from each universe (a single copy of the object might exist in each universe).
Policy P6	*Type: Acquire Miss Policy* A replica of the object is created on a node if at least 20% of total request for the object have been denied to this node.
Policy P7	*Type: Acquire Miss Policy* A replica of the object is created on a node if at least 40% of total requests for the object have been denied to this node.
Policy P8	*Type: Object Update Policy* A replica of the object is created on a node if the object has been updated in at least 20% of total requests issued from each node.
Policy P9	*Type: Object Update Policy* A replica of the object is created on a node if the object has been updated in at least 40% of total requests issued from each node.

Table 1 summarizes the tested replication policies. Using these policies we were able to show how the GUN replication engine reacts to various request patterns. For each policy the average acquire time for each type of object was recorded. Table 2 presents the acquire time for the *BuildingRules* object for all nine replication policies. As this object is accessed by all worker processes, they greatly benefit if the object is replicated closer to them. From Table 2 it can be seen that the acquire time for all workers is lower when using replication for all replication policies. This can also be observed in Figure 9. When object replication is not employed the acquire time is lower only for worker processes running in the universe where *BuildingRules* object was created.

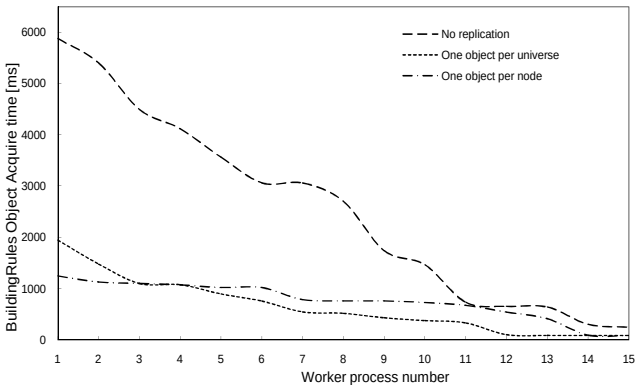

Figure 9 *BuildingRules acquire time*

Table 2 *Average Acquire time for the BuildingRules object*

Policy	P1	P2	P3	P4	P5
Acquire time [ms]	2536.93	652.47	761.00	1812.93	2143.40
Policy	P6	P7	P8	P9	
Acquire time [ms]	1397.33	1867.93	1560.8	1797.13	

The influence of the policy type on the acquire time can be observed in Figure 10 which compares the acquire time for the Object Update Policy applied at universe level and at node level. If we focus on similar policies for the two replication levels (e.g. the pairs P4-P8 and P5-P9) we can conclude that replicating at node level results in lower acquire time than when replicating at universe level. By looking at the differences between the values for acquire time for the two types of policies, we can say that when the

number of worker processes in a universe that use the same object is large, it is better to replicate the object on each worker node, but if the number of workers is rather small using a universe level policy can still assure good performance with a lower storage capacity usage.

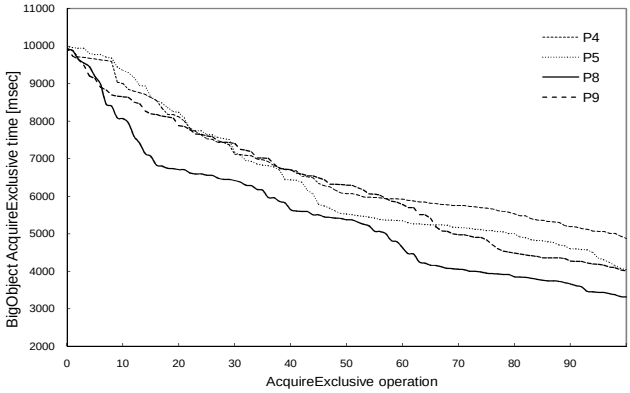

Figure 10 *Node vs.Universe update*

The time when replication takes place has effects on the time required to acquire an object exclusively or non-exclusively. In Figure 10 for P8 replication occurs early, after just 20% of total updates, and thus the acquire time drops faster than for P9 where replication occurs after 40% of total updates. Same thing can be observed in Figure 11, for the Acquire Miss Policy.

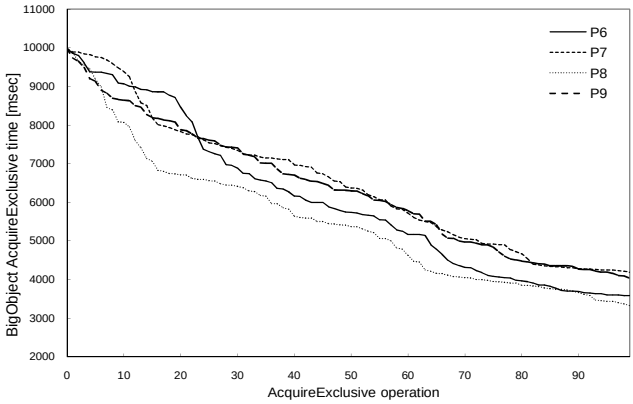

Figure 11 *Object Update vs. Acquire Miss policies*

Table 3 *Acquire exclusive success rate*

Policy	*Success rate [%]*
No replication	87.88
One object per universe	90.94
One object per node	95.30

Although the acquire success rate for all policies was approximately equal, we noticed differences in acquire exclusive success rate. These differences are shown in Table 3. By creating a replica of an object on each node that uses the object, acquire exclusive success rate can be increased with approximately 10% compared with the case when a single copy of the object exists in the grid universe. This is because through replication the time necessary for writing the object decreases and as a consequence the time the object is locked by a client decreases and therefore more clients can be served over a shorter period of time.

8 CONCLUSION AND FUTURE WORK

In this paper we have presented an overview of the success criteria for a grid shared data service. We have explored both traditional system aspects and new aspects that tend to be overlooked in several system designs and papers. We have briefly presented some of the most important achievements in the emerging domain of grid shared data programming and highlighted some of their drawbacks.

We have introduced a new idea on designing grid shared data services, based on our observations on past and current attempts. We have described the basic design strategies in system separation, replication handling, consistency specification and mutual exclusion. We have emphasized the original idea of combining memory consistency specification with type coherence in an object oriented model and argued on the importance of thorough system verification on a wide scale and extreme conditions.

Last but not least, we have introduced the architecture of a java based prototype called GUN that implements the abstract model of the proposed system. We have extended the GUN prototype with a generic replication engine. Finally we have shown the results of replication related experiments that show promising results in terms of both performance as well as quality parameters such as improved access success rates.

As part of our future work we focus on extending the GUN experiments in order to highlight different aspects to those presented in this paper. More important, we aim to conduct a computer aided performance analysis on our model using the PRISM probabilistic model checker (Kwiatkowska, Norman and Parker, 2002).

REFERENCES

Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S. and Foster, I. (2001) 'Secure, Effcient Data Transport and Replica Management for High-Performance Data-Intensive Computing', *In Proceedings of the 18th IEEE Symposium on Mass Storage Systems (MSS 2001)*, Large Scale Storage in the Web, page 13, Washington, DC, USA, IEEE Computer Society.

Antoniu, G., Bougé, L. and Mathieu, J. (2005) 'JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid', *Scalable Computing: Practice and Experience*, Volume 6, Pp. 45-55.

Baduel, L., Baude, F. and Caromel, D. (2002) 'Efficient, Flexible and Typed Group Communications for Java', *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, pp. 28-36, ISBN 1-58113-559-8.

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.

Bershad, B., Zekauskas, M., and Sawdon, W. (1993) 'The Midway distributed shared memory system', *In Proceedings IEEE COMPCON Conference*, IEEE, pp 528-37.

Bertier, M., Arantes, L. and Sens, P. (2004) 'Hierarchical token based mutual exclusion algorithms', *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, ISBN: 0-7803-8430-x, Page 539- 546.

Caron, E. and Desprez, F. (2006) 'DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid', *International Journal of High Performance Computing Applications*, 20(3), Pp. 335-352.

Carson, M. and Santay, D. (2003) 'NIST Net: a Linux-based network emulation tool', *SIGCOMM Computer Communication Review*, 33(3):111–126.

Foster, I. and Kesselman, C. (1998) 'The Grid: Blueprint for a New Computing Infrastructure', Morgan Kaufmann Publishers.

Foster, I. and Kesselman, C. (1997) 'Globus: A Metacomputing Infrastructure Toolkit', *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115-128.

Karasavvas, K., Antonioletti, M., Atkinson, M., Hong, N., Sugden, T., Hume, A., Jackson, M., Krause, A., and Palansuriya, C. (2005) 'Introduction to OGSA-DAI Services', *Lecture Notes in Computer Science*, pp. 1-12, Volume 3458.

Karonis, N.T., Toonen, B. and Foster, I. (2002) 'MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface'.

Kwiatkowska, M., Norman, G., and Parker, D. (2002) 'PRISM: Probabilistic Symbolic Model Checker', *Proceedings of TOOLS 2002*, volume 2324 of Lecture Notes in Computer Science, pages 200-204, Springer.

Lamport, L. (1979) 'How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs', *IEEE Transactions Computers*, Vol. C-28, No. 9, pp. 690-1

Lee, C., Matsuoka, S., Talia, D., Sussman, A., Mueller, M., Allen, G. and Saltz, J. (2001) 'A Grid Programming Primer', *Tech report*, Advanced Programming Models Research Group.

Macariu, G., Tudor, D. and Cretu, V. (2008) 'Designing a dynamic replication engine for grid shared data programming', *SYNASC International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, in print.

Naimi, M., Trehel, M., and Arnold, A. (1996) 'A log (N) distributed mutual exclusion algorithm based on path reversal', JPDC, 34(1), pp. 1–13.

Osrael, J., Froihofer, L. and Goeschka, K.M. (2006) 'A Replication Model for Trading Data Integrity against Availability', *The 12th Int. Symp. on Pacific Rim Dependable Computing (PRDC'2006)*, IEEE CS Press.

Seigneur, J.M., Biegel, G. and Damsgaard, C. (2003) 'P2P with JXTA-Java pipes', *Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, Ireland.

Sopena, J., Legond-Aubry, F., Arantes, L. and Sens, P. (2007) '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.

Tudor, D., Cretu V., and Schreiner, W. (2008), 'Designing an architecture for distributed shared data on the grid', *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, volume 22 of Lecture Notes in Computer Science, pages 261–264.