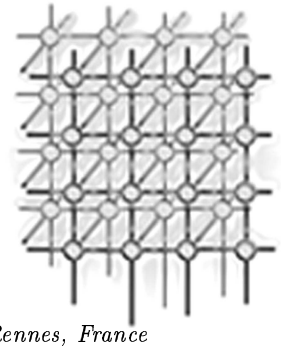
How to bring together fault tolerance and data consistency to enable grid data sharing

Gabriel Antoniu, Jean-François Deverge, Sébastien Monnet

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

How to bring together fault tolerance and data consistency to enable grid data sharing



G. Antoniu*,†, J.-F. Deverge and S. Monnet

IRISA/INRIA and University of Rennes 1, Campus de Beaulieu, 35042 Rennes, France

SUMMARY

This paper addresses the challenge of transparent data sharing within computing grids built as cluster federations. On such platforms, the availability of storage resources may change in a dynamic way, often due to hardware failures. We focus on the problem of handling the consistency of replicated data in the presence of failures. We propose a software architecture which decouples consistency management from fault tolerance management. We illustrate this architecture with a case study showing how to design a consistency protocol using fault-tolerant building blocks. As a proof of concept, we describe a prototype implementation of this protocol within JuxMem, a software experimental platform for grid data sharing, and we report on a preliminary experimental evaluation of the proposed approach.

KEY WORDS: grid computing, data sharing, fault tolerance, consistency protocols

1. Introduction

Data management in grid environments is currently a topic of major interest to the grid computing community. However, as of today, no approach has been widely established for transparent data sharing on grid infrastructures. Currently, the most widely-used approach to data management for distributed grid computation relies on *explicit data transfers* between clients and computing servers. As an example, the Globus [12] platform provides data access

*Correspondence to: Gabriel Antoniu, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes, France
†E-mail: Gabriel.Antoniu@irisa.fr
Contract/grant sponsor: GDS Project of French ACI MD initiative and the Brittany Region

mechanisms based on the GridFTP protocol [1]. Though this protocol provides authentication, parallel transfers, checkpoint/restart mechanisms, etc., it is still a transfer protocol which requires *explicit* data localization. On top of GridFTP, Globus integrates data catalogs [1], where multiple copies of same data can be manually registered. The consistency of these replicas are however at the user's charge. IBP [6] provides a large-scale data storage system, consisting of a set of buffers distributed over Internet. The user can "rent" these storage areas and use them as temporary buffers for efficient data transfers across a wide-area network. Transfer management is still at the user's charge and no consistency mechanisms are provided for the management of multiple copies of the same data. Finally, Stork [17] is another recent example of system providing mechanisms to *explicitly* locate, move and according to the needs of a sequence of computations. It proposes an integrated approach allowing the user to schedule data placement just like computational jobs. Again, data location and transfer are at the user's charge.

Within the context of a growing number of applications using large amounts of distributed data, we claim that *explicit management of data locations* by the programmer arises as a major limitation against the efficient use of modern, large-scale computational grids. Such a low-level approach makes grid programming extremely hard to manage. In contrast, the concept of *data sharing service* for grid computing [3] has been proposed, with the goal to provide *transparent access to data*. This approach is illustrated by the JuxMem software experimental platform. The user only accesses data via a global identifier. The service handles data localization and transfer without any help from the programmer. However, it is able to use additional hints provided by the programmer, if any. The service also transparently uses adequate replication strategies and consistency protocols to ensure data availability and consistency. These mechanisms target a large-scale, dynamic grid architecture. In particular, the service supports events such as storage resources joining and leaving, or unexpectedly failing. This is the framework within which we conducted the study presented in this paper.

Problem: keep replicated data consistent. The goal of a data-sharing service is to allow grid applications to access data in a distributed environment. We are considering scientific applications, typically exhibiting a code-coupling scheme: e.g. multiple weakly-coupled codes running on different sites and cooperating via periodical data exchanges. In such applications, shared data are *mutable*: they can be read, but also *updated* by the different codes. When accessed on multiple sites, data are often replicated to enhance access locality. Replication is equally used for fault tolerance, since grid nodes may crash. To ensure that read operations do not return obsolete data, consistency guarantees have to be provided by the data service. These guarantees are defined via *consistency models* and are implemented using *consistency protocols*.

Difficulty: handling consistency in a dynamic context. The problem of sharing mutable data in distributed environments has intensively been studied during the past 15 years within the context of Distributed Shared Memory (DSM) systems [18, 20]. These systems provide transparent data sharing, via a unique address space accessible to physically distributed machines. When the nodes modify the data, some consistency action is triggered (e.g.,

invalidation or update), according to some consistency protocol. A large variety of DSM consistency models and protocols [7, 13, 15, 20, 24] have been defined, their role being to specify which remote nodes have to be notified of the modification, and when. They provide various trade-offs between the strength of the consistency guarantees and the efficiency of the implementation.

However, traditional DSM systems have generally demonstrated satisfactory efficiency (i.e., near-linear speedups) only on *small-scale* configurations: in practice, up to a few tens of nodes [20]. This is often due to the intrinsic lack of scalability of the algorithms used to handle data consistency. Most of the time, they rely on global invalidations or global updates of all existing data copies. On the other hand, an overwhelming majority of protocols assume a *static* configuration where nodes do not disconnect nor fail. It is clear that these assumptions do not hold any more in the context of a *large-scale*, *dynamic* grid infrastructure. Faults are no longer exceptions, but they become part of the general rule; resources may become unavailable and eventually become available again; finally, new resources can dynamically join the infrastructure. In such a context, consistency protocols cannot rely any more on entities supposed to be stable, as traditionally was the case. A new approach to their design is definitely necessary, to integrate these new hypotheses.

This idea is at the core of the design of the grid data-sharing service we introduced in [3]. The service is defined as a hybrid system inspired by DSM systems (for transparent access to data and consistency management) and P2P systems (for their scalability and volatility-tolerance). This paper makes a further step by proposing an approach allowing *consistency protocols* to take into account *fault tolerance* through decoupled management of these two aspects. The motivations and the general principles are presented in Section 2. In Section 3 we describe the detailed architecture and we show how to use traditional group communication components of fault-tolerant distributed systems [11, 19] as building blocks for consistency protocols. The approach is illustrated in Section 4 with a case study explaining the design of a fault-tolerant consistency protocol. Section 6 shows how this protocol has been implemented in the JuxMem platform and presents a preliminary experimental evaluation. Some concluding remarks and future directions are given in Section 7.

2. Approach: decoupling fault tolerance management from consistency management

Let us first note that both fault tolerance mechanisms and consistency protocols are traditionally implemented using *replication*. However, the underlying motivations are totally different for each of the two uses.

Replication in consistency protocols. Consistency protocols use data replication for *performance* issues, to allow multiple nodes to read the same data in parallel via local accesses. However, when a node modifies a data copy, the consistency protocol is activated, e.g. the other copies must be updated or invalidated, to prevent subsequent read operations from returning

invalid data. Note that P2P systems also use replication to enhance access locality, but most of them do not address consistency issues, since data is generally immutable.

Replication for fault tolerance. Replication is also commonly used by fault tolerance mechanisms [22] to enhance *availability* in an environment with failures. When a node hosting a data copy crashes, other copies can be made available by other nodes. Various replication strategies have been studied [14], leading to various trade-offs between efficiency and the level of fault tolerance guaranteed.

In distributed systems where both consistency *and* fault tolerance need to be handled, replication can be used with a double goal. Consequently, depending on whether these two issues are addressed separately or not, two architectural designs are possible.

Integrated design. A possible approach consists in addressing consistency and fault tolerance at the same time, relying on the same set of data replicas. For instance, data copies created by the consistency protocols to enhance data locality can serve as backup if crashes occur. Conversely, backup replicas created for fault tolerance can be used by the consistency protocol. This approach has a major disadvantage: the design of the corresponding software layer is very complex, as illustrated by some fault-tolerant DSM systems [16, 23].

Decoupled design. A different approach consists in designing the consistency protocol and the fault tolerance mechanism separately. This approach has several features. First, the design of consistency protocols is simplified, since the protocols do not have to address fault tolerance issues at a low level. Therefore, it is possible to leverage existing consistency protocols. Only some limited interaction between the consistency protocol and the fault tolerance mechanism needs to be defined (see Section 3.2). Second, consistency protocols and fault tolerance strategies can be developed independently. This favors a cleaner design, each of the two components being dedicated to its specific role. Finally, this approach provides the ability to experiment multiple possibilities to couple various consistency protocols with various fault tolerance strategies.

The goal of this paper is to discuss how to manage consistency and fault tolerance at the same time, in a *decoupled* way, using this second approach.

3. Building consistency protocols based on fault-tolerant components

In general, traditional consistency protocols for DSM systems rely on stable entities in order to guarantee that data accesses are correctly satisfied. For instance, a large number of protocols associate to each data a node holding the most recent data copy. This is true for the very first protocols for sequential consistency [18], but also for recent *home-based* protocols implementing lazy release consistency [24] or scope consistency [15], where a *home node* is in charge of

maintaining a reference data copy. It is important to note that these protocols implicitly assume that the home node never fails. Such an assumption cannot be made in a dynamic grid environment, where faults may occur. In such a context, it is important to avoid such *single points of failure*, whose crash would compromise the behavior of the whole system. Therefore, the role of home node has to be played by an entity able to transparently react to faults and disconnections, in order to maintain a given degree of availability for the reference data copy.

Our proposal is to enhance the availability of such entities by using some basic building blocks that have been defined within the context of fault-tolerant distributed systems [11, 19]: replication mechanisms, group membership protocols, atomic multicast, consensus, etc. We introduce these blocks in next section. Then, we describe the "glue layers" through which the consistency protocol interacts with these fault-tolerant blocks.

3.1. Fault-tolerant components: a short overview

Failure model. We are considering two types of failures that need to be addressed in a grid environment. First, nodes may crash, i.e. nodes act normally (receive and send messages according to their specification) until they fail (crash failures). This failure model is known as the *fail-stop* model. Second, we assume messages can be delayed or lost, due to buffer overflows or to temporary link failures. We assume *fair-lossy* communication channels. If a process p sends a message m to another process q an infinite number of times through a fair-lossy channel, and if q does not fail, then q eventually receives m from p. Informally, we assume that network links may duplicate or lose some messages, but not all of them.

In our failure model, we consider two main timing aspects: the communication delays and the computation times. We make the assumption that upper bounds upon these times exist but are not known. Thus our algorithms assume an asynchronous timing model, using a *failure detection* mechanism. Such a service is in charge of providing a list of nodes suspected to have failed. Classical fault tolerance mechanisms are often built on these hypotheses, which are realistic in a grid context.

Basic abstractions. Based on the hypotheses mentioned above, a number of abstractions have been defined for the management of different aspects related to fault tolerance in distributed systems.

Group membership protocols . The *group membership* abstraction [11] is a mechanism providing the ability to manage a set of nodes having a common interest. The nodes belonging to a group have to store the current composition of the group (i.e. the member list). As nodes may join or leave the group and even crash, the member lists are changing. The *group membership* protocol has to ensure a certain degree of consistency of these lists by synchronizing the members views of the group. Between two view synchronizations, the same set of messages should be delivered by all the nodes within a group. In our case, the *group membership* mechanism applies to a group of nodes that play together the role of a *home* entity of consistency protocols.

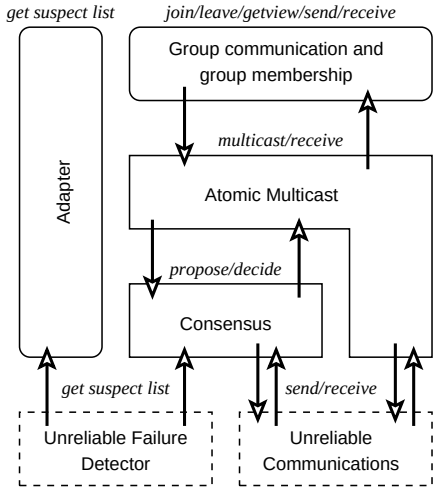


Figure 1. *An architecture for group communication and group membership protocols.*

Atomic multicast. The home entity is in charge of maintaining the reference data copy. It is represented by a group of nodes on which the reference data copy is replicated. As in our model nodes may crash, to ensure that an up-to-date copy will remain available we use a pessimistic replication mechanism. Therefore, with our replication scheme, all the replicas are updated simultaneously. This goal can be achieved by delivering all messages in the same order to all group members. Members of the group have to agree upon an order for message delivery and this agreement is reached using standard *consensus* protocols.

Consensus protocols. A *consensus* protocol allows a set of nodes to agree on a common value: each node proposes some value and the protocol ensures that (1) eventually all nodes that do not fail decide a value, (2) that value has been proposed by some node and (3) the decided value is the same for all unfaulty nodes. In our case the decision is about the order in which messages are delivered to the group members. The consensus problem in asynchronous systems can be solved thanks to *unreliable failure detectors* [10]. The role of these detectors is to provide a list of nodes suspected to be faulty. The consensus protocol can cope with the approximate accuracy of the list contents.

These blocks can interact with each other in many ways. In this paper, we consider a layered, decoupled design (Figure 1), inspired by [19]. Here, the *adapter* module allows higher-level software layers to register to the failure detection service and to filter the list of suspected nodes according to some user-specified quality of service, as in [8].

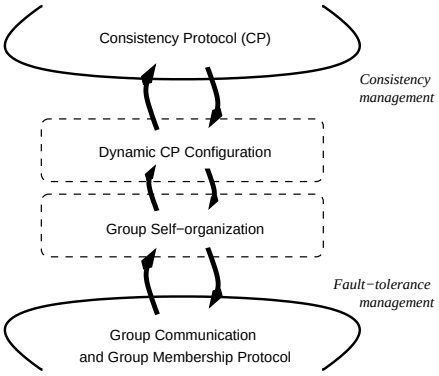


Figure 2. *Decoupled architecture for managing consistency and fault tolerance.*

3.2. Using fault-tolerant components in consistency protocols

Our idea is to use the abstractions described above to build fault-tolerant entities able to play the role of critical entities in consistency protocols. For instance, each *home node* can be replaced by a group of nodes handled via a group membership interface and supporting atomic multicast. However, some actions like *1) group self-organization* or *2) configuration of new group members* need to be handled by higher-level layers. Such actions are not necessarily specific to consistency protocols (i.e. they can apply to several consistency protocols). They are situated precisely at the "boundary" between fault tolerance management and consistency management. Hence, we need to introduce two interface layers in our architecture, as shown in Figure 2.

Group Self-organization. This layer handles the composition of a group of nodes that together act as a home node, by enriching the semantics of the traditional *group membership* abstraction by including a *group membership policy*. The layer decides when to remove from the group nodes reported to be faulty, by parameterizing the QoS of the failure detector. It also removes nodes that notify about their future disconnections. Following such removals, the layer adds new members to the group, to maintain the availability of the home node. To do so, it takes into account constraints specified at allocation time: the necessary memory size, the network performance, or the replication policy (expressed in terms of number of clusters where to spread data replicas, number of replicas per cluster, etc.). Various trade-offs could be expressed at this level (e.g. smaller group sizes to enhance communication efficiency vs. larger group sizes to increase the level of fault tolerance).

Dynamic Consistency Protocol Configuration. When some new node is added to the group that acts as a home node, the newcomer has to initialize his state in order to be consistent with the state of the other members of the group. The *Dynamic Consistency Protocol Configuration* layer defines how to instantiate a consistency protocol on such nodes. The new node must first take into account the configuration messages generated by the other members of the group at the level of this layer, before reacting to external messages addressed to the group.

In the decoupled architecture we propose, the *Group Self-organization* layer and the *Dynamic Consistency Protocol Configuration* layer set up a slim interface through which the consistency protocol interacts with the fault tolerance strategy. Thus, each of these entities can be designed independently according to its specific goals, and only a limited interaction needs to be defined between them.

4. Case study: designing a hierarchical, fault-tolerant consistency protocol

The typical grid applications we target are loosely code-coupling applications, in which several codes run in parallel on different clusters and iteratively exchange data. These data exchanges can be carried out through read or write accesses to a data-sharing service, such as JuxMem [3]. The role of this service is to ensure consistent access to shared data, while transparently handling failures. This is where fault-tolerant consistency protocols relying on the approach proposed in Section 3 are useful. To illustrate this idea, this section describes how to build such a protocol starting from a *non fault-tolerant* protocol implementing the *entry consistency model*. We first introduce the entry consistency model and a basic, non fault-tolerant protocol which implements it. We then show how this protocol can be made fault-tolerant using the approach proposed in the previous section.

4.1. A non fault-tolerant consistency protocol for the entry consistency model

Previous experience with DSM consistency protocols has shown that relaxed consistency models can be implemented via efficient protocols at the price of restricted consistency guarantees. For instance, the programmer must use synchronization operations, such as `acquire`, to make sure the subsequent accesses are correctly satisfied, and `release`, to allow the local modifications to be (eagerly or lazily) propagated to remote nodes. This general requirement is valid for models like release consistency [13], entry consistency [7] or scope consistency [15].

In this paper, we focus on the *entry consistency model*. As opposed to other relaxed models, it requires an explicit association of data to synchronization objects. This allows the model to leverage the relationship between a synchronization object that protects a critical section, and the data accessed within that section. A node's view of some data becomes up-to-date

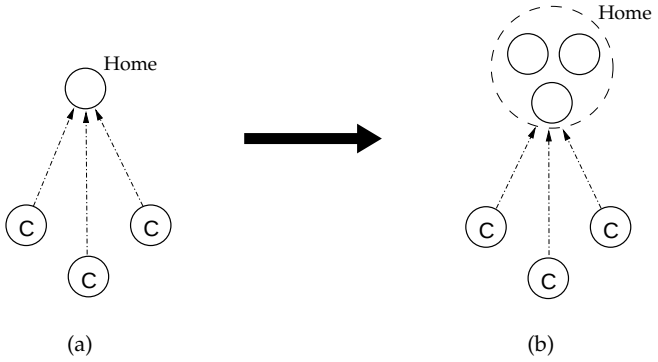


Figure 3. *Building a fault-tolerant consistency protocol.*

only when the node enters the associated critical section. This eliminates unnecessary traffic, since only nodes that declare their intention to access data will get updated, and only the data which will be accessed will be updated. Such a concern for efficiency makes this model a good candidate in the context of scientific grid computing.

The programmer has to observe two main requirements. First, all shared data have to be associated with at least one guarding synchronization object. Second, exclusive accesses to shared data have to be explicitly distinguished from non-exclusive accesses by using two different primitives: `acquire`, which grants mutual exclusion; `acquireRead`, which allows non-exclusive accesses on multiple nodes to be performed in parallel. A detailed description of the model is given in [7].

In this case study, our starting point is a *non fault-tolerant* protocol for entry consistency (Figure 3(a)). We are considering a *home-based* protocol, in which a *home node* is associated to each data. This node is responsible for maintaining a reference copy for that data. The home node also manages a lock associated to its data. When a process enters a critical section protected by such a lock, the associated shared data is updated on the node hosting that process (if necessary). On leaving the critical section, the local modifications (if any) are transmitted to the home node. Consequently, accesses to shared data involve some communications with the home node.

4.2. Deriving a *fault-tolerant* protocol

In the protocol sketched out above, the home node is clearly a critical entity that must be available for the protocol to be operational. Since in a grid environment we cannot realistically assume that such entities will be implemented by failure-free nodes, this is where the approach proposed in Section 3 can be applied. Our proposal is to make these entities fault-tolerant by using an enriched version of the *group membership* abstraction. The home node is replaced by a group of nodes (Figure 3 (b)). This group of nodes has the following properties: 1)

All messages sent to such a group are received *by all members of the group, in the same order* (atomic multicast); 2) The groups are self-organizing: they maintain some user-specified replication degree by dynamically and adding new members when necessary in a "smart" way. The selection of the new members is handled by the *Group Self-organization* layer, whereas their initialization is managed by the *Dynamic Consistency Protocol Configuration* layer, as explained in Section 3.2.

The number of simultaneous faults supported by this solution depends on the implementation of the underlying fault-tolerant building blocks (consensus, atomic broadcast). Our current implementation supports up to $\lfloor \frac{n-1}{2} \rfloor$ simultaneous failures within a group, where n is the group size.

Note that the consistency protocol can use the new *home* entity, composed of multiple nodes, exactly as it initially used the home node in the original, non fault-tolerant version. It still assumes the home is always available, but this property is now achieved transparently for the protocol, thanks to the implementation of the Self-organizing Group Membership abstraction. Thanks to this approach, the consistency protocol implements *exactly the same distributed algorithm* as in its initial, non fault-tolerant version. The consistency protocol and the replication-based fault tolerance mechanism are thus clearly decoupled.

4.3. Going large-scale: a hierarchical, fault-tolerant protocol

Let us note that, in a grid consisting of clusters federation, inter-cluster latency is generally higher then intra-cluster latency. In order to improve the protocol efficiency, a suitable approach can rely on minimizing the inter-cluster communications. This idea has been used in some DSM systems and has lead to the design of *hierarchical* consistency protocols. In CLRC [5], local caches are created on each cluster, to optimize the locality of consecutive accesses to remote data modifications. In [4], this approach is applied to distributed lock management, by reordering lock requests: requests from the local cluster are served before remote requests.

Let us now consider a *hierarchical* version of the protocol sketched out in the Section 4.1. This version, illustrated on Figure 4(a), is inspired by the hierarchical, home-based protocol for release consistency described in [4]. The idea is to use a *two-level hierarchy of home nodes.* On each cluster, a *local home* will serve accesses from the local cluster, whereas a *global home* will serve data accesses to the clusters, i.e. to the local homes. When a client needs to access some data, it will require the associated lock to its local home. If this home owns the corresponding access rights to the data, it can satisfy the access. Otherwise, it will request the lock from the global home, with an updated copy of the data. Note that the global home only serves the requests issued by the local homes; it has no control on what requests are subsequently served by the local homes. However, to minimize inter-cluster communications, a local home serves local requests with higher priority than remote requests issued on other clusters, received via the global home. To avoid starvation, a limit is set on the number of consecutive accesses served by each local home, so that remote requests be served too.

The next step is to make this hierarchical protocol fault-tolerant. To this purpose, we use the same technique described in the previous section. We replace each local home by a group

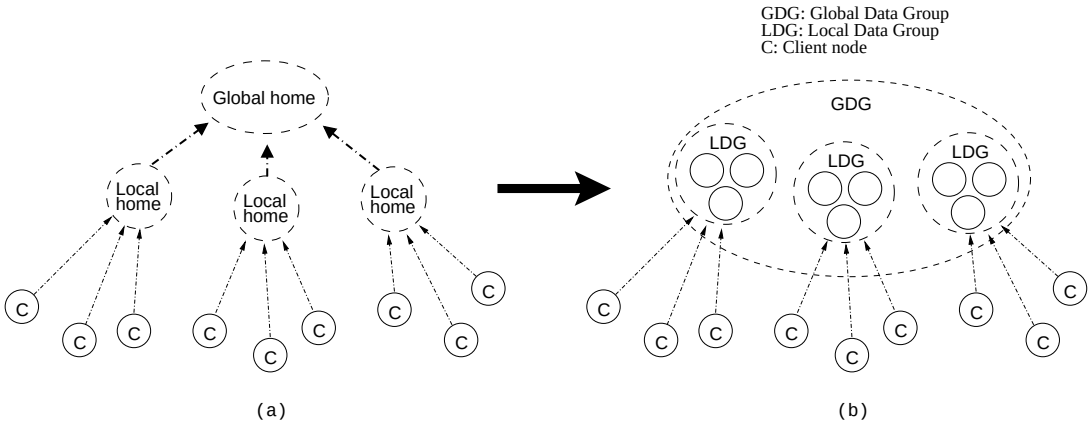


Figure 4. *Building a hierarchical, fault-tolerant consistency protocol.*

of nodes, that we call *Local Data Group* (LDG). At a higher level, the global home is replaced by a *Global Data Group* (GDG), whose members are the LDGs (Figure 4 (b)). The GDG and the LDGs have the self-organizing properties as detailed in the previous section: they maintain some user-specified replication degree by dynamically and adding new members when necessary.

5. Limits of the proposed approach and possible extensions

The central idea of the proposed approach consists in using replication and group communication abstractions in order to enhance the availability of critical protocol entities. This way, the consistency protocol could transparently tolerate two kinds of failures: 1) crashes of the nodes that implement, as a group, the critical entity; 2) temporary failures of the communication links between these nodes.

5.1. Coping with client failures

The approach could be extended by using the same technique for enhancing the availability of other entities involved in the protocol. For instance, the clients accessing the data could also be replicated. However, depending on the application, this is not always possible. The unavailability of sensors (hardware timers) or application deployment obstacles (software environment dependencies, security policies, software license restrictions, etc.), may make it impossible to replicate the client. In such cases, for applications where it is important to tolerate client failures, different techniques have to be used.

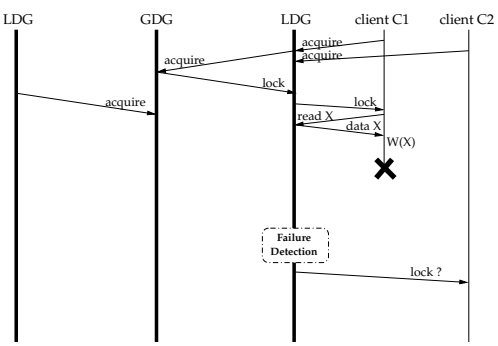


Figure 5. Failure of the exclusive lock owner

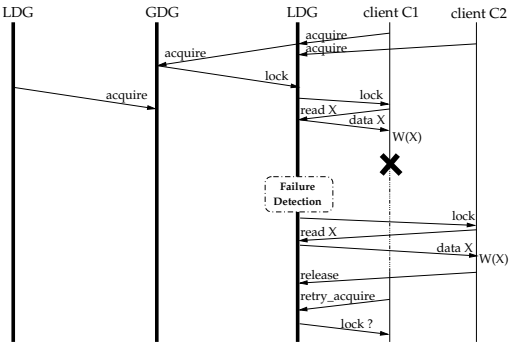


Figure 6. Error of the LDG's failure detector

Let us consider a situation (illustrated on Figure 5), where a client $C1$ holding some lock crashes. To ensure the liveness of the locking mechanism, the lock manager (here: the LDG) could decide to *force* the lock release when it detects the client's failure. This way, other clients having requested the lock (e.g. $C2$) would be able to acquire it. However, the correctness of such a scheme is dependent on the potential impact of the actions performed by the faulty client during its critical section on the actions that the other clients waiting for the lock are supposed to perform. Here are a few possible situations.

Let us suppose, for instance, that the client $C1$ modifies only some local data until it crashes and that these modifications are not reflected in the globally shared data. In this case, the global data is still valid and the client $C2$ could decide to ignore $C1$'s failure and proceed with the execution of its critical section. This situation corresponds, for instance, to our consistency protocol, which does not perform any global action before the end of the critical section. The modifications performed in a critical section are globally propagated by the release operation.

Let us now assume that the actions performed by the client $C1$ have some global impact and that some globally shared data remain inconsistent if $C1$ fails before finishing its critical section. In this second case, $C2$ cannot ignore $C1$'s failure, otherwise it will process inconsistent data.

Another situation that may lead to inconsistencies is related to false failure detections (remember that the failure detector is assumed to be unreliable). On Figure 6, the LDG's failure detector wrongly decides that the client $C1$ fails. This may be caused by some badly-configured timeout, or by the slowdown of this client, due to some temporary overload. The LDG then decides to force the lock release and lets client $C2$ to acquire the lock and modify some shared data. Later, client $C1$ tries to release the lock, but it realizes that it has already been released by force. In this case, a solution may be to try to acquire the lock again and re-execute the critical section.

In all these cases, it is clearly important for the lock primitives (`acquire`/`release`) to return the user some information about the possible failures that may have been detected, so that the user may make the right decision: either ignore the failure, or trigger some rollback/recovery actions. Such actions are generally application-dependent and should not be handled at the level of the distributed synchronization mechanism. We are currently working on an enhanced mutual exclusion mechanism, able to track client failures and report relevant information about the failure history.

5.2. Coping with failures of cluster-level data groups

In our hierarchical scheme, the situations described above for client-level failures may also be generalized for cluster-level failures (remember that the LDGs behave as clients of the GDG). A LDG can tolerate a limited number of simultaneous failures of the nodes that compose it. If more faults occur, or if the whole underlying physical cluster is down, the LDG fails. Here again, there may be two main situations.

If the failure occurs while some local client has modified the data protected by that lock, then the situation is similar to the one described in the previous subsection and may generally be handled in a similar manner.

If the whole underlying cluster is down, or if no client on that cluster holds the lock at the time of the failure, then there will be no data inconsistencies with other clusters. Let us now assume that a majority of the nodes that make up a LDG on some cluster L fail, while some client C accesses the data on that cluster. The failure event is detected by the GDG, which forces the lock release, in order to satisfy lock requests issues by other clusters. The client C may trigger the instantiation of a new LDG on cluster $L1$. It can then discover that the data has been concurrently modified on cluster L (by C itself) and on other clusters. In such a situation application-level recovery mechanisms are necessary, similar to the ones mentioned in the previous section.

5.3. Tuning the group replication level

Another important aspect of our proposed approach is related to the replication degree to be used for each group entity. Consistency protocols rely on transparent replica management by the group entities. However, the consistency protocol user may want to tune the replication level in order to obtain a good tradeoff between performance and fault tolerance. The *Group Self-organization* layer has to implement some replica management policy, like node selection for the replacement of faulty nodes. This is illustrated on Figure 7, where the self-organizing group membership protocol chooses node D to replace the faulty node C.

Protocol policies could point out nodes with certain properties. *Quantitative policies* can optimize a tradeoff between fault tolerance and performance by carefully managing the group size (i.e. the replication degree). Hence, a group made of many nodes will support many concurrent node failures, but it will waste memory space and will lower the efficiency of data

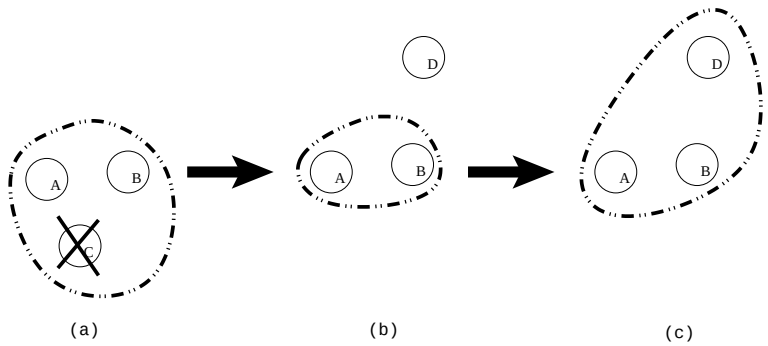


Figure 7. *Self-organizing group membership protocols have to replace faulty replicas.*

updates. Conversely, fewer nodes will improve the communication performance but will yield lower fault tolerance. Currently, the replication degree is set up explicitly by the user at data allocation time. An interesting feature would be to let the user specify some application-level semantics (e.g. the *data criticity*), and then let the system automate the selection of the most appropriate replication degree, also by taking into account the reliability of the physical infrastructure.

On the other side, *qualitative policies* could be used in order to select nodes for their characteristics. As an example, a policy may find nodes with low churn rates to increase group resiliency. On the other hand, node sets with high bandwidth and low latency for internal communication could automatically be defined, in order to enhance group communications performances. As a downside, these policies would require more complex informations on the underlying physical environment.

6. Implementation and preliminary evaluation

To experiment our approach, we have used the JuxMem software experimental platform for grid data sharing, described in [3]. We have refined its architecture according to the decoupled approach proposed in this paper and we have implemented the fault-tolerant consistency protocol described in Section 4.2.

The general architecture of JuxMem mirrors a federation of distributed clusters and is therefore *hierarchical* (Figure 8). It consists of node sets, called `cluster` groups, which correspond to physical clusters. These groups are included in a wider group, the `juxmem` group, which gathers all the nodes running the data-sharing service. Note that these *service groups* consist of different nodes with different states. They do not make up a replicated service and

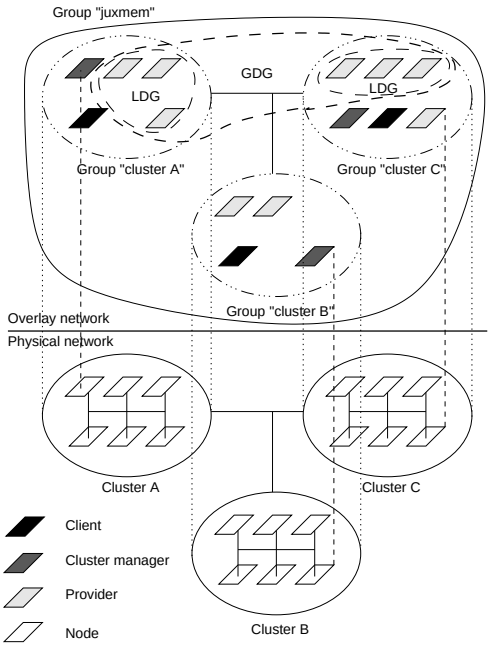


Figure 8. JuxMem: *a hierarchical architecture for a grid data service.*

do not rely on the same abstractions (group membership, atomic broadcast) as the groups previously described, that act as home nodes. Any `cluster` group consists of *provider* nodes which supply memory for data storage. The memory available in the group is handled by a *cluster manager*. Any node (including providers and cluster managers) may use the service to allocate, read or write data as *clients*, in a peer-to-peer approach. This architecture has been implemented using the JXTA [25] generic P2P platform.

When allocating memory, the client has to specify on how many clusters the data should be replicated, and on how many nodes in each cluster. This results into the instantiation of the GDG and LDG entities used by the consistency protocol, as explained in Section 4.2. In the example shown on Figure 8, data is replicated across two LDGs created on two different clusters. Each LDG is made up of three physical nodes. The allocation operation returns a global data ID. To read/write a data block, clients only need to specify this ID. The platform transparently locates the corresponding local LDG or instantiates it if necessary. Subsequent accesses to data are directed to this LDG by the consistency protocol.

At the low level of our architecture, the LDG and GDG components have been implemented based on the fault-tolerant, leader-based group communication protocol proposed in [9]. Our

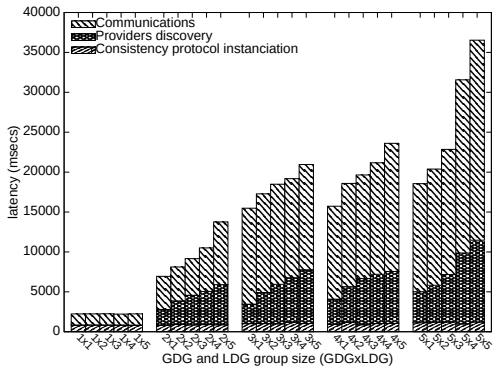


Figure 9. *Allocation costs depends on replication degree.*

implementation supports node crashes and link failures. In each LDG or GDG group, up to $\lfloor \frac{n-1}{2} \rfloor$ failures are supported, where n is the group size.

Preliminary evaluation. For our preliminary experiments, we have used the JDF [2] deployment suite to run our tests over a 64-node cluster of 2,4 GHz bi-Pentium IV with 1 GB RAM, interconnected through a Fast-Ethernet network. We have partitioned our physical cluster into 8 `cluster` groups, 8 nodes each. In order to emulate a cluster federation, Dummynet [21] has been used to add latency between the `cluster` groups. The average latency between any two nodes that belong to different groups has been set to 15 ms, a typical value for long distance networks, whereas the intra-cluster latency is 0.25 ms. Our software environment is JuxMem running over JXTA 2.2.1 and Java 1.4.2.

We first analyzed the impact of the replication degree on the cost of data allocation. The allocation procedure consists of 3 steps: 1) the client has to discover enough providers in the JuxMem network to satisfy the replication degree; 2) the client sends allocation requests to a set of discovered providers, selected in order to satisfy the user-specified constraints (concerning replication degrees, locality, etc.); 3) the selected providers perform the actual allocation and instantiate the consistency protocol layer and the necessary group communication components; this results in creating the corresponding LDGs and GDG.

We have evaluated the impact of the replication degree on the allocation cost by varying the sizes of the GDG and LDG groups (Figure 9). We can note that: 1) the architecture initialization cost is largely overcome by the communication involved by the first two steps described above (discovery and allocation requests); 2) the discovery cost grows linearly with respect to the replication degree; 3) the cost of the actual allocation is quasi-constant despite the number of required replicas, because the client makes all these requests in parallel.

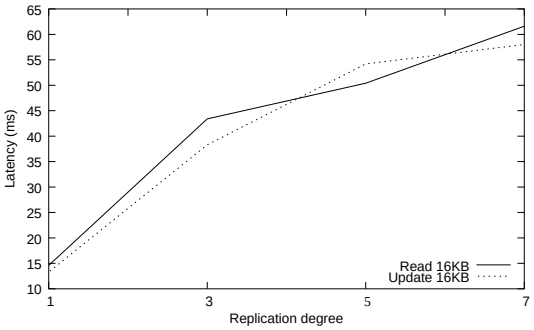


Figure 10. Cost of the basic primitives - 16 KB

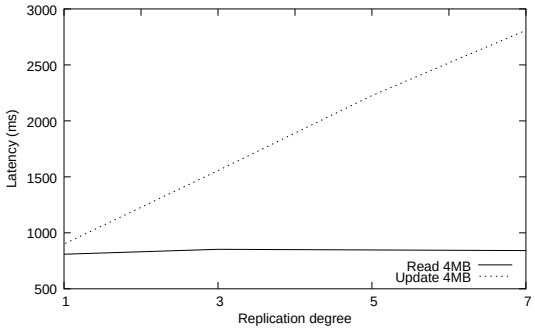


Figure 11. Cost of the basic primitives - 4 MB

We have also measured the cost of the basic operations of the consistency protocol: data read and data update. These operations involve communications between a client and its local LDG. We measured the cost of these operations while varying the cluster-level replication degree (i.e. the LDG size). This is illustrated on Figures 10 and 11. First, we can note that the overhead due to replication is significant for small data sizes (e.g. 16 KB): the read and update operations are three times slower, because our atomic multicast protocol uses a two-phase commit strategy. However, this cost increases very slowly with the replication degree. Second, for large data sizes (e.g. 4 MB), the fault tolerance overhead is negligible compared to the data transfer delay. The cost of update operations linearly increases with the replication degree. This is due to our leader-based implementation of the group communication protocol, where the leader node sends the data to all the group members across the network.

Further planned measurements will evaluate the service throughput while one client performs writes and another perform reads (i.e. producer/consumer scheme). We also plan an experimental study the impact of failures on the performance of the service operations.

7. Conclusion

In this paper, we have addressed the problem of handling the consistency of replicated data in a grid data-sharing service. In such a context, the availability of storage resources changes dynamically. We have shown the advantages of a software architecture which decouples consistency management from fault tolerance management. We have illustrated our approach by showing how to design a fault-tolerant consistency protocol which implements the entry consistency model. As a preliminary experimental validation, we have implemented a prototype of the proposed fault-tolerant consistency protocol within JuxMem, a software experimental platform for grid data sharing.

The main advantage of the proposed approach is that it allows the consistency protocol and the replication strategy to be designed independently, while only a small interaction has to be defined through the *Group Self-organization* and the *Dynamic Consistency Protocol Configuration* layers. Thereby, existing consistency protocols can be made fault-tolerant by carefully defining this interaction. Different trade-offs (e.g., efficiency vs. level of fault tolerance) can be obtained by tuning this interface. Such studies are part of our planned future work.

The policy implemented by the *Group Self-organization* layer should become adaptive (e.g. by varying the replication degree) using a *monitoring* module. If this policy is well-tuned in order to fit the characteristics of the physical architecture, the availability of the home nodes will be guaranteed most of the time. This is true as long as the assumptions made about the fault types and about the number of concurrent faults are correct. Otherwise, recovery will not be possible, and the user application will be informed about this by the consistency protocol. It is then its responsibility to react, according to its specific constraints (retry, rollback, etc.). Such events should however be extremely rare if the self-organizing group membership policy is correctly tuned. We are currently working on extensions of our approach, in order to define an extended semantics of the consistency protocol, which should take into account such cases.

REFERENCES

1. William Allcock, Joseph Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.
2. Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in P2P experiments using the JXTA distributed framework. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par)*, volume 3149 of *LNCS*, pages 1038–1047, Pisa, Italy, August 2004. Springer.
3. Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. In *Proceedings of the 1st Workshop on Adaptive Grid Middleware (AGridM)*, pages 49–59, New Orleans, Louisiana, September 2003.
4. Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM consistency protocol hierarchy-aware: An efficient synchronization scheme. In *Proceedings of the 3rd IEEE/ACM International Conference on Cluster Computing an the Grid (CCGrid)*, pages 516–523, Tokyo, Japan, May 2003. IEEE.
5. Luciana Bezerra Arantes, Pierre Sens, and Bertil Folliot. An effective logical cache for a clustered LRC-based DSM system. *Cluster Computing Journal*, 5(1):19–31, January 2002.
6. Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A study in resource sharing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 194–201, Berlin, Germany, May 2002. IEEE.
7. Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON)*, pages 528–537, Los Alamitos, CA, February 1993.
8. Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 354–363, Washington, DC, June 2002.
9. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.
10. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
11. Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

12. Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
13. Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 15–26, Seattle, WA, June 1990.
14. Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.
15. Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 277–287, Padova, Italy, June 1996.
16. Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 289–298, Pasadena, California, June 1995.
17. Tevfik Kosar and Miron Livny. Stork: Making data placement a first-class citizen in the grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 342–349, Tokyo, Japan, March 2004.
18. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
19. Sergio Mena, André Schiper, and Pawel Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of the 4th International Middleware Conference (Middleware)*, volume 2672 of *LNCS*, pages 414–432, Rio de Janeiro, Brazil, June 2003. Springer.
20. Jelica Protić, Milo Tomasević, and Veljko Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE, August 1997.
21. Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.
22. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
23. Florin Sultan, Thu Nguyen, and Liviu Iftode. Scalable fault-tolerant distributed shared memory. In *Proceedings of the IEEE/ACM Supercomputing (SC)*, pages 54–55, Dallas, Texas, November 2000.
24. Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Seattle, WA, October 1996.
25. The JXTA project. `http://www.jxta.org/`.