# A Fault-Tolerant Transparent Data Sharing Service for the Grid

Louis Rilling, Christine Morin

# *A Fault-Tolerant Transparent Data Sharing Service for the Grid*

Louis Rilling, Christine Morin

**N°5427**

Décembre 2004

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5427--FR+ENG

# A Fault-Tolerant Transparent Data Sharing Service for the Grid

Louis Rilling*, Christine Morin*

Systèmes communicants
Projet Paris

Rapport de recherche n˚5427 — Décembre 2004 — 26 pages

**Abstract:** We consider a transparent data sharing service for distributed applications in the Grid. Our service may alleviate the burden of the user and the programmer to manage the distribution and the migration of data by transparently locating, caching, and managing the consistency of the data. To fit in a large scale and dynamic environment such as the Grid, our data sharing service tolerates every reconfiguration (benign failures, arrivals and departures of sites) in the system's life time, provided that no more than a fixed number of reconfigurations occur simultaneously. This service relies on application backward error recovery and replication to ensure the liveness of the application. Reconfiguration has a high impact on data location mechanisms. To solve this issue, our service uses a distributed hash table provided by a structured overlay network. We have experimentally evaluated our service and present in this paper an analysis of the results.

**Key-words:** Grid Computing, Fault-Tolerance, Transparent Distributed Data Sharing, Consistency

*(Résumé : tsvp)*

* {Louis.Rilling}{Christine.Morin}@irisa.fr

# Partage de données transparent et tolérant aux fautes pour la grille

**Résumé :** Nous nous intéressons à un service de partage transparent de données pour les applications distribuées de la grille. Notre service allège le fardeau de l'utilisateur et du programmeur, consistant à gérer la distribution et le transfert des données, en localisant, mettant en cache, et gérant la cohérence des données de manière transparente. Pour être adapté à un système à grande échelle comme la grille, notre service de partage de données tolère toutes reconfigurations (défaillances bénignes, arrivées et départs de sites) dans la vie du système, à condition qu'au plus un nombre fixé de reconfigurations aient lieu simultanément. Ce service utilise des recouvrements en arrière d'applications et de la duplication pour assurer la vivacité de l'application. Les reconfigurations ont un impact élevé sur les mécanismes de localisation des données. Pour résoudre ce problème, notre service utilise une table de hachage distribuée fournie par un réseau logique structuré. Nous avons évalué expérimentalement notre service et nous présentons dans cet article une analyse des résultats.

**Mots-clé :** Grilles de calcul, tolérance aux fautes, partage transparent de données distribuées, cohérence

## 1 Introduction

In computational grids, applications handle large amounts of data that are distributed on the executing sites. Locating, caching, and managing the transfer of these data are complex tasks left to the programmer. We consider a transparent data sharing service to ease these complex tasks.

The number of sites and their geographical distribution make the system dynamic: at any time, several sites may decide to join the system, leave the system, can be disconnected from the system due to a network failure, or can fail. In the paper, we call such events *reconfiguration events*. However, the sites are not supposed to behave maliciously. A data sharing service for the grid should handle these reconfiguration events to help the user to execute distributed applications reliably despite reconfiguration.

### 1.1 Transparent Data Sharing Service

We consider applications consisting of several (parallel) components, each executing on one node (a site), loosely coupled using the data sharing service (DSS). On behalf of the components, the DSS uses a consistency protocol to transparently locate, transfer, and cache the data. We chose the model of atomic consistency (named *coherence* in the paper) for its simplicity: this is the model implemented in shared memory multiprocessors, and propagating the modifications of the data needs no explicit synchronization.

The main issues to design a transparent data sharing service are the scalability of the consistency protocol and the inherent dynamicity of the system. In this paper, we partly consider the scalability issue in the model of system we assume: several reconfiguration events can occur simultaneously, and messages can be delivered in disorder and several times. However we mainly focus on supporting dynamic reconfiguration of the system.

The DSS we present tolerates reconfiguration events. The DSS prevents the application from blocking despite *every* reconfiguration events in the system's life time, provided that no more than a fixed number of reconfiguration events occur simultaneously. This DSS can use application checkpoint and restart mechanisms to automatically restart the application when one component of the application is lost (node failure or network failure), or when the application is blocked because a shared data is lost. The data sharing service reduces the cases in which shared data can be lost, having the data located most of the time on a node executing a component of the application. These properties make our DSS interesting for a dynamic distributed system like a computational grid.

### 1.2 Related Work

Several projects currently study data sharing in wide area distributed systems as a global, persistent, data store service. InterWeave [9] allows the programmer to choose the level of consistency for each shared data in order to optimize the transfers of data, but does neither provide fault tolerance nor support for dynamic reconfiguration. JuxMem [1] aims at alleviating the task of the programmer to manage data in the grid. Using a P2P design, JuxMem also aims at providing fault tolerance and support for dynamic reconfiguration of the system. JuxMem uses a dynamic group communication system to replicate the data and manage the consistency of the replicas [3].

Other research has been done to extend Software Distributed Shared Memory (SDSM) protocols to federated clusters, by making the protocol hierarchy aware [4, 2]. However none of them consider fault tolerance neither dynamic reconfiguration of the system.

Although reliably executing shared memory applications on distributed systems has been well studied, most of these systems were static clusters of workstations with at most a few dozens of nodes [18]. In particular, the proposed fault-tolerant distributed shared memory systems tolerate only one node failure, and must reconfigure themselves before they can tolerate another failure [26, 12]. Moreover, only one node can be removed or added per reconfiguration. In contrast, the sites of a computational grid should be able to join or leave the system at will. Thus it is important to tolerate several *simultaneous* reconfiguration events.

In peer-to-peer like distributed systems, data sharing has mostly been studied as file storage. Recent works provide file systems ensuring from NFS-like to sequential consistency of data [20]. Files are distributed and replicated over the computers (or nodes) of the system in order to both speed up accesses and ensure the availability of shared data despite the intermittent availability of the nodes [23]. Among these systems, only OceanStore provides consistency management for caches. However OceanStore gives no strong guarantees on the consistency: the user has to explicitly check that the retrieved data is the latest version [20].

We present in this paper an evaluation of the coherence protocol of our DSS. This protocol has already been presented and proved in [21]. The protocol improves previous works on SDSM systems with higher fault-tolerance. The protocol tolerates every reconfiguration event in the system's life time, provided that no more than $f$ reconfiguration events occur in a same reconfiguration, $f$ being a parameter of the protocol. We outline the protocol in Section 2. In Section 3, we give a model of the system that clarifies the model described in [21]. In order to be complete, we recall some important details of the protocol, including the main proofs, in Sections 4 and 5. We present in Section 6 an experimental evaluation of the protocol in our data sharing service that shows its practical interest for dynamic distributed systems. Finally, Section 7 concludes the paper.

## 2 Overview of the Coherence Protocol

Our data sharing service acts as a cache of shared objects. The components of the application access only their local copies. For example, the components can access the shared objects through their address space. To simplify the presentation, we assume in this section that only one object is shared. This section outlines the protocol. We detail the protocol in Sections 4 and 5.

We consider atomic consistency protocols based on the write-invalidate method. The main other method for atomic consistency is write-multicast, which can be implemented using for example atomic multicast, or quorum-acknowledged multicast [15]. Write-invalidate protocols propagate updates lazily whereas write-multicast protocols propagate updates eagerly. The write-multicast method has the drawback to catch every write operation in order to update the other copies. This is too costly if shared data should be transparently accessed using the virtual memory mechanisms of the processors [14]. Moreover every component updates its copy, even when it does not use it.

Write-multicast protocols have the advantage to actively replicate the shared data, which allows executions to progress without rollback when less than a majority of components crashes. In our case however, unless the application is specifically designed to tolerate crashes of components without needing to rollback, and unless active replication is required, using checkpointing allows to adjust the cost of fault-tolerance for each use.

First we discuss our contribution. Second we introduce the entities of the protocol, and finally show how the protocol supports dynamic reconfiguration of the system.

### 2.1 Contribution and Discussion on the Model of System

Our protocol brings two contributions. First the model of system suits better large scale distributed systems: messages can be delivered in disorder and several times. This is important because keeping the messages in order or suppressing duplicates forces a node to maintain during all its connection time a state about every node it has ever communicated with. Such book-keeping definitely lowers the scalability of the system. To our knowledge, no consistency protocol has considered this scalability issue.

Second, our protocol suits *dynamic* distributed systems: it tolerates every reconfiguration events in the system's life time, provided that no more than $f$ reconfiguration events occur in a same reconfiguration window, $f$ being a parameter of the protocol. A reconfiguration window is informally defined as the time needed for the protocol and the system to reconfigure after a reconfiguration event. To our knowledge, no other write-invalidate protocol tolerates several simultaneous reconfiguration events.

### 2.2 Entities of the Protocol

The protocol is similar to the write-invalidate-based coherence protocol of K. Li that uses fixed distributed managers [14]. The entities of the protocol are: the *nodes* of the system, which can have *copies* of the shared object; the *object manager*, which handles nodes' requests for copies so that coherence is preserved; and the *application manager*, which manages components and helps the object manager to ensure liveness. Assigning copies to nodes instead of components allows components to migrate transparently with respect to the coherence protocol.

There is always exactly one *master copy*, which is the only writable copy. The location of the master copy changes according to write accesses from the components. The node having the master copy is called the *owner*.

The object manager acts as a directory service by recording the set of the nodes having a valid copy. The object manager does not keep any copy of the object.

The object manager and the application manager run on nodes. However these managers can move to other nodes when reconfiguration events occur (see paragraphs 2.3 and 5.2).

Figures 1 and 2 show how the entities interact, on the example of a node requesting a writable copy. A request for a write access consists of 4 phases. First the requesting node sends a WRITE request to the object manager. Second, the object manager requests the node to confirm its WRITE request (WRITE-CONFIRM request), and the requesting node confirms (WRITE-CONFIRM message). Third, the object manager invalidates the other valid copies (INVALIDATE requests), and the nodes concerned acknowledge the invalidation (INVALIDATE-ACK messages). Fourth, the object manager forwards the WRITE request to the owner, which sends a copy (COPY message) to the requesting node.

Similarly, a request for a read access consists of 2 phases. First the requesting node sends a READ request to the object manager. Second, the object manager forwards the request to the owner, which sends a copy (COPY message) to the requesting node.

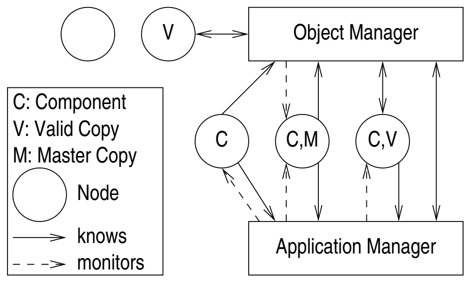

Figure 1: Organization of the Entities

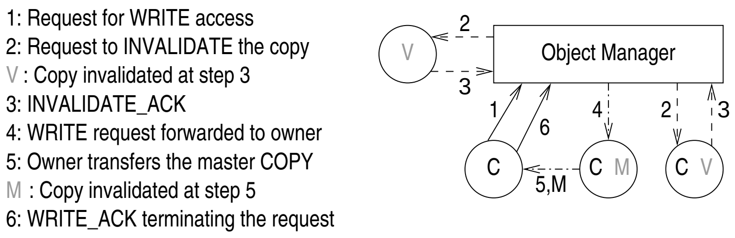
1: Request for WRITE access
2: Request to INVALIDATE the copy
V : Copy invalidated at step 3
3: INVALIDATE_ACK
4: WRITE request forwarded to owner
5: Owner transfers the master COPY
M : Copy invalidated at step 5
6: WRITE_ACK terminating the request
V
2
3
Object Manager
1
6
4
2
3
C
C M
C V
5,M

Figure 2: Messages sent to request write access to a copy. The messages' names appear in upper case characters.

## 2.3 Transparently Locate the Managers

Our protocol solves part of the issues raised by dynamic reconfiguration by transparently locating object managers and application managers using a distributed hash table (DHT). Indeed, reconfiguration is difficult to handle in earlier work mainly because managers, or other equivalent directory structures, are partly identified by the nodes on which they execute. As a result, each time a node disconnects, (i) the structures have to be transferred to some other nodes, and (ii) the other nodes of the system have either to be informed of the new locations of the structures, or to apply deterministic rules to determine the new locations.

Our protocol identifies managers as keys in a DHT. The DHT service transparently solves part (ii) of the changes of location, even in the case of multiple simultaneous reconfiguration events. As a result, the size of a reconfiguration window depends on the reconfiguration window of the DHT service. To solve part (i), we use state machine replication (see paragraph 5.2.1).

Distributed hash tables, which recent peer-to-peer systems provide [22], solve the location issue using a structured overlay network on top of the distributed system. Each node in the overlay is a computing site and is given an identifier (ID). Each node maintains a set of logical neighbors. For example, these neighbors are the nodes having the closest ID to the node's ID. Each node is responsible for a subset of the key space, which can be determined knowing only the IDs of the logical neighbors. For example, keys have identifiers similar to those of nodes and a key is mapped to the node having the closest ID. Keys are accessed by routing messages through the overlay network.

DHTs have good properties in dynamic distributed systems. If a node joins or leaves the system, the changes in the mapping of keys to nodes are local, that is restricted to the logical neighborhood of the joining (resp. leaving) node. In particular, only the nodes in this neighborhood have to be notified. Most existing consistency protocols that are not write multicast-based[1] change the locations of *all* object managers at each reconfiguration [26].

Moreover, the DHT can be tuned to keep the mapping consistent despite up to $f$ simultaneous reconfiguration events. To our knowledge, no existing consistency protocol that is not write multicast-based tolerates several simultaneous reconfigurations.

## 2.4 Fault Tolerance Issues

In order to tolerate (unexpected) disconnections, the protocol ensures two properties: (i) the valid copies of the shared object are always coherent (safety), (ii) the execution of the application is never indefinitely blocked (liveness). To maintain both properties, first we adapted K. Li's protocol to support reordering and duplicated messages, and to ensure the safety property. Second we made the protocol ensure the liveness property. We prove in paragraph 5.3 that the resulting protocol ensures the safety property despite any number of simultaneous reconfiguration events, and the liveness property despite up to $f$ simultaneous reconfiguration events.

We ensure liveness using checkpoint and restart of the application, and state machine replication for the managers. Obviously, unneeded restarts should be avoided. Therefore we ensure (as surely as failure detectors allow) that, at each time $t$, the only cases that need to restart the application are failures of nodes executing at least one component of the application at time $t$. Hence, assuming that failure detectors are perfect during a major part of the system's life time, restarts of the application are mostly triggered when no other means can make the execution progress.

To limit the number of restarts, we prevent any dependence on nodes on which no component of the application is running. Thus, each time a node $n$ ceases to execute components for the application, node $n$ ensures that it has no copy of the shared object. If node $n$ is not the owner, the copy is silently discarded. Otherwise, the master copy is injected to another node executing at least one component.

## 2.5 Other Impacts of Reconfiguration

The previous paragraph has shown how our protocol tolerates failures. The other reconfiguration events are joins and leaves (that is voluntary disconnections) of nodes.

To tolerate joins, our protocol only has to ensure that the managers remain reachable through the DHT. This concerns only joins of nodes whose IDs are close to the ID of a manager. To that end, we replicate the managers on the neighborhood of the nodes they are mapped to. This is the technique used for fault-tolerant DHT storage [23].

If a node $n$ voluntarily disconnects, our protocol has first to ensure that managers are sufficiently replicated to tolerate $f$ other reconfiguration events. This concerns only leaves of nodes whose IDs are close to the ID of a manager. Second, if node $n$ executes components of the application, node $n$ makes these components migrate to other nodes[2], and prunes dependences of the application on itself by injecting the master copies of shared objects to other nodes (for example, to the nodes the components are migrated to). Figure 3 summarizes the impact of a leave.

Table 1 summarizes the impact of reconfiguration events.

[1]Write multicast-based protocols do not need any object manager since all the copies know the locations of each other.

[2]The policy used to decide the locations of components is assumed to be implemented in another entity that we call resource manager, whose description is beyond the scope of this paper.

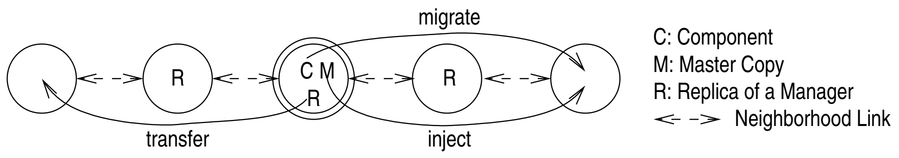

Figure 3: Actions to take when the circled node voluntarily disconnects

| **Case for the node** | **Join** | **Voluntary Disconnection** | **Failure** |
|---|---|---|---|
| Executes no component | nothing | nothing | nothing |
| Executes at least one component | nothing | migrate & inject | restart |
| Node ID close to a manager's key | transfer a replica | transfer a replica | create a replica |
| Node ID far from any manager's key | nothing | nothing | nothing |

Table 1: Cases and actions to handle a reconfiguration event from a node, with respect to components (upper half), and replicas of managers (lower half)

## 3 System Structure and Model

In this section, we describe the model of distributed system we adopted to design our coherence protocol.

### 3.1 Crash/Recovery Model

#### 3.1.1 Nodes

To represent the intermittent availability of the nodes in the system, we assume a crash/recovery model [13] for the nodes. The nodes alternate between periods in which they are *up* and periods in which they are *down*. While a node is *up* it behaves according to its specification. A node *disconnects* (*crashes* in [13]) when it is *up* and becomes *down*. A node disconnects when either it voluntarily leaves the system or it fails by stopping every computation and every communication. While a node is *down*, it does not communicate with any other node. A node *joins* (*recovers* in [13]) when it is *down* and becomes *up*.

When a node recovers, its state is clean and empty: it does not include anything about the applications it has executed in the previous *up* periods. Unlike other works on the crash/recovery model [13], we do not consider stable storage for the nodes. As a result, we assume that when a node voluntarily leaves the overlay, it will discard all its state before joining the system again.

When a node joins the system, it introduces itself with a unique identifier $I$ (for example a static IP address) and its connection time $t_c$. This time stamp can be generated using the node's local clock. The pair $(I, t_c)$ identifies an *instance* of a node. All instances of a node are unique in the system's life time.

We also assume that as soon as a node $n_1$ connected to the system learns about another node $n_2$ also connected, $n_1$ also learns the instance $(I, t_c)$ of node $n_2$.

#### 3.1.2 Replicated Entities

We say that an entity replicated over several nodes *crashes* when a node hosting a replica crashes and no other replica exists. The replicated entities of our protocol are the object managers and the application managers (see paragraph 5.2). Since nodes recover with an empty state, replicated entities do not recover.

### 3.2 Model of Communications

The nodes of the system communicate by sending messages through point-to-point bidirectional channels. Like in [13] and [11], we assume that messages can be delivered in disorder and several times.

Messages can take arbitrary long times to reach their destination, and processors can be arbitrary slow. This makes the system *asynchronous*.

We consider fair lossy channels: if a node sends a message an infinite number of times to a same node, the message is delivered an infinite number of times unless the receiver disconnects. Moreover, we assume that if a message is routed through the overlay network (see paragraph 2.3) an infinite number of times towards a same key of the DHT, it is also delivered an infinite number of times to the target key unless the key crashes.

However our protocol does not handle directly message loss. We assume that an entity (node or replicated entity) that has sent a message will retransmit the message until it either receives an acknowledgment from the target entity, or it disconnects or crashes. For messages routed to keys, smarter solutions to prevent message loss in structured overlays have been studied in [16].

Since nodes and replicated entities (in particular keys) can crash, some messages may never be delivered despite retransmission. As illustrated in Figure 4, the only messages that can get lost are those that are not received before the sender (node $n_0$) crashes (message m3), and those that may have been received but not processed by the target (node $n_1$) before it crashes (messages m4 and m6).

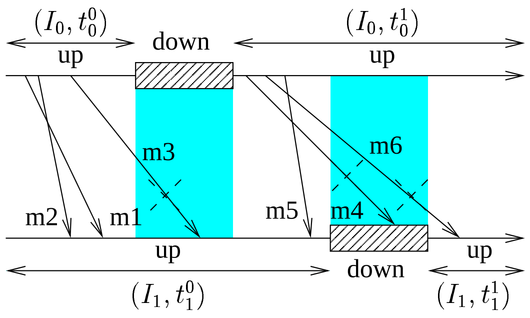

Figure 4: Only the messages crossing the shaded areas can be lost.

As a result, when a message is sent, then either it will be delivered, or one of the sender and the receiver disconnects or crashes. Because of asynchrony and messages delivered in disorder, no other distinction than learning that the message has been delivered can be made between these three possibilities.

## 3.3 Reduction to the Asynchronous Crash/No-Recovery Model with Failure Detectors

Unlike [13] we consider a crash/recovery model without stable storage, but with unique instances of nodes. As a result, our model can also be considered as a dynamic (nodes can join the system at any time) crash/no-recovery model for *instances* of nodes: once a node has disconnected, it never joins under the same instance again. We say that an instance *disconnects* when the represented node disconnects.

For this reason, our protocol considers only instances of nodes. As a result, a message sent to a node in instance $(I, t)$ is not delivered to this node in any instance $(I, t')$ with $t' \neq t$ (message m6 in Figure 4). In the remainder of the paper, unless it is stated otherwise, *an instance of a node will be abusively referred to as a node*.

We augment the asynchronous model with failure detectors targeted towards the crash/no-recovery model. Instead of detecting failures of nodes, these detectors detect failures of *instances* of nodes: once a node $n$ crashes under an instance $i$, the failure detectors that monitor $i$ should consider node $n$ as down, even after node $n$ rejoins (under a different instance).

Nodes dispose of unreliable failure detectors allowing to detect if a specified (instance of a) node has disconnected. Failure detectors achieve strong completeness: if a node $n$ disconnects, there is a time after which the failure detector *suspects* node $n$.

A node in a specified group of nodes is *correct* if it does not disconnect as long as it belongs to the group. We also assume that the nodes in a specified group of nodes dispose of failure detectors in class $\diamond S$ [8], which achieve strong completeness and eventual weak accuracy: there is a time after which some correct node in the group is not suspected by any node in the group as long as it belongs to the group.

### 3.4 Virtual Global Real Time Clock

We assume the existence of a virtual global real time clock. Although *no node* can access this clock, such a clock is required to define atomic consistency.

### 3.5 Model of Applications

The applications we consider are distributed applications made up of several components that share a state using the data sharing service. Shared objects are private to an application. In order to keep the focus on the data sharing aspect concerning checkpoint and restart for fault tolerance (see paragraph 5.1.1), we assume that a component consists in a single thread of execution that communicates with the other components using shared objects only. We also assume that external inputs and outputs can be replayed without loosing any input nor corrupting any output (for example, a file can be reread, and writes only occur at specified positions in a file).

## 4 Basic Coherence Protocol

In this section, we describe a protocol that ensures safety (coherence of the copies), but not liveness (that is the application may block indefinitely) if reconfiguration events occur. We show how we handle reconfiguration events to ensure liveness in Section 5. We assume without loss of generality that there is only one shared object.

Our basic protocol adapts an asynchronous version of K. Li's protocol based on fixed distributed manager [14] to messages delivered in disorder and several times. To our knowledge, no asynchronous version of this protocol has been implemented, even assuming FIFO reliable channels.

The intrinsic scheme of write-invalidate protocols and our crash/recovery model make these adaptations sufficient to obtain a protocol that ensures safety despite any number of simultaneous reconfiguration events. This intermediate protocol also ensures liveness if no reconfiguration event occur and if failure detectors are perfect.

We chose to handle messages delivered in disorder and several times at the protocol level instead of the message passing layer. Otherwise each node would need to keep information about every node that has communicated at least once with it, which is definitely not scalable. Our protocol needs only that the object manager keeps a list of nodes that have a valid copy of the object. Moreover, since the object manager is replicated for fault tolerance (see paragraph 5.2), duplicated messages are generated at the protocol level and hence must be handled at this level.

We quickly present an asynchronous version of K. Li's protocol in paragraph 4.1. We present our adaptation to messages delivered in disorder and several times in paragraph 4.2.

### 4.1 Write-Invalidate Protocol Based on Fixed Distributed Managers Assuming FIFO Reliable Channels

#### 4.1.1 Nodes

Nodes reclaim access rights on their copies when an access fault (READ fault or WRITE fault) occurs. Hence, for a node, the local copy can be in one of four main states: INVALID when no right to access, SHARED when read-only access, OWNER_SHARED for a master copy with read-only access, and OWNER_EXCLUSIVE for a master copy with read-write access. In state OWNER_EXCLUSIVE, all other copies are in state INVALID. At any time, exactly one copy is a master copy. The other valid copies are cached copies.

Before the copy of a node $n$ enters state SHARED or state OWNER_EXCLUSIVE, node $n$ must retrieve an up to date copy from the owner. To that end, node $n$ routes a READ request (respectively WRITE request) to the object manager. The reply to a READ request is a COPY message returned by the owner. The reply to a

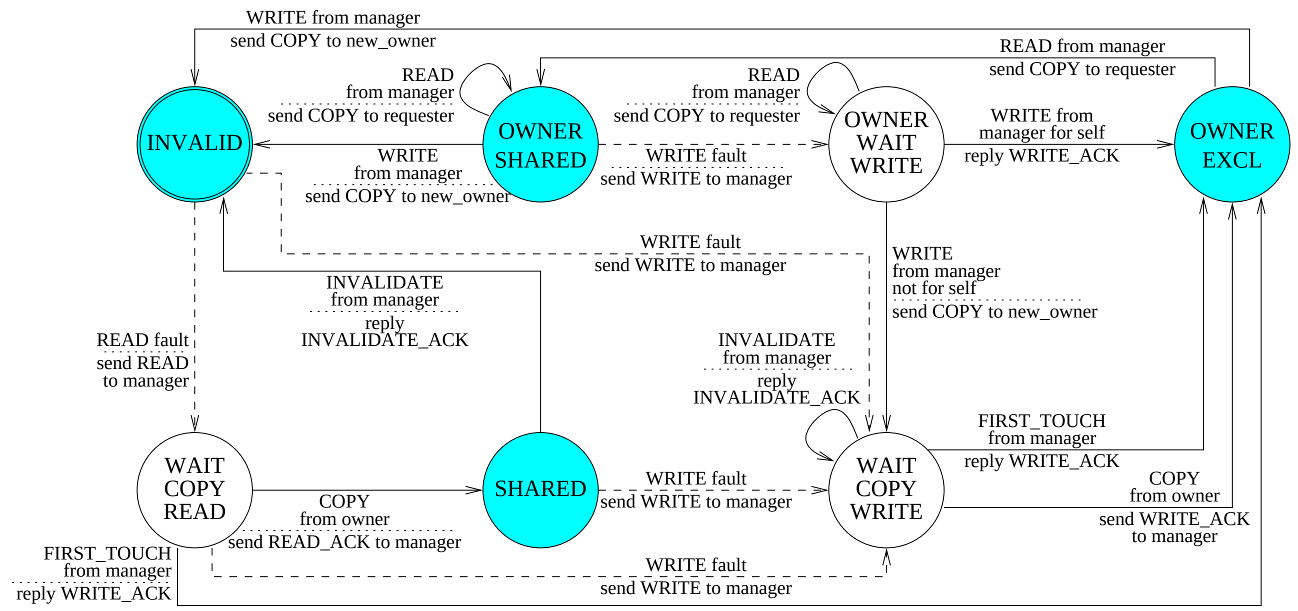

Figure 5: Asynchronous State Machine of a Node with FIFO Reliable Channels. The dashed lines represent the transitions that send requests. These transitions are triggered by local access faults. The main states of the protocol are represented by filled circles. The dotted lines separate the conditions to apply the transitions from the messages sent in consequence.

WRITE request is a COPY message from the owner if node $n$ is not the owner, or the request itself that has been redirected by the manager if node $n$ is already the owner.

The first node $n$ that requests a copy gets in reply a FIRST-TOUCH message from the manager, which means that node $n$ must create the object and become the owner.

### 4.1.2 Object Manager

The object manager keeps track of the owner's ID, and redirects nodes' requests to the owner. It also ensures that before it redirects any WRITE request, all other copies are in state INVALID. To that end, it maintains a *copy set* containing the IDs of all the nodes but the owner having a valid copy.

### 4.1.3 Asynchronous Input/Output State Machines

The behaviors of nodes and object managers are described using deterministic input/output state machines. Similarly to the write-invalidate protocol based on *dynamic* distributed managers (recall that our protocol is based on *fixed* distributed managers) implemented in DSM-Threads [19], both state machines are asynchronous: message receptions occur only before transitions. Besides the benefits in responsiveness and for multi-threading (see [19]), this is required for active replication in our protocol (see paragraph 5.2). As a consequence, the node's state machine includes more states in which the protocol waits for the reception of a copy with the desired access rights (states WAIT_COPY_READ and WAIT_COPY_WRITE), or for the manager's authorization to write to the master copy (state OWNER_WAIT_WRITE). Figures 5 and 6 show the state machines respectively for a node and the object manager (OM). For readability purposes, only the conditions to apply the transitions (above the dotted lines) and the messages sent in consequence (below the dotted lines) are shown in the figures.

In all states, the OM queues nodes' requests (WRITE and READ requests) in pending_req and handles them using a FIFO policy. This ensures that each request will be satisfied within a finite delay. In state IDLE, the OM just waits until the queue contains a request.

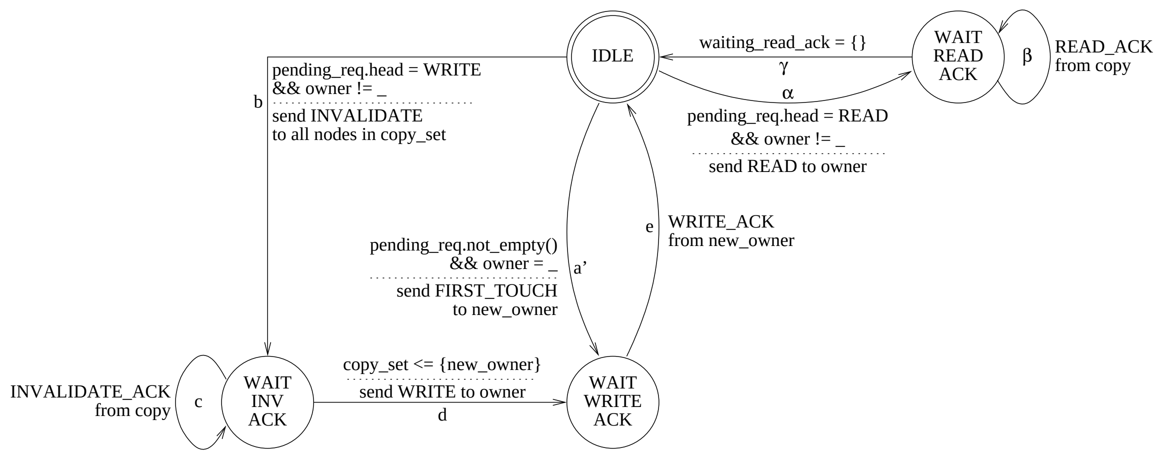

Figure 6: Asynchronous Object Manager Assuming FIFO Reliable Channels. The dotted lines separate the conditions to apply the transitions from the messages sent in consequence.

The requests are terminated by acknowledgments (READ_ACK and WRITE_ACK messages) that ensure that WRITE requests are handled atomically. This protects the liveness of the application (requests are effectively redirected to the owner and hence do not get lost) and the coherence of the copies.

READ requests are handled as much as possible simultaneously (according to the FIFO policy of the queue) by entering state WAIT_READ_ACK. waiting_read_ack contains the IDs of the nodes from which READ requests have been redirected to the owner (node owner) but not acknowledged yet in transition $\beta$.

## 4.2 Adaptation to Messages Delivered in Disorder and Several Times

Our protocol's state machines, which are designed to support messages delivered in disorder and several times, are represented in figures 7 and 8. The bodies of the transitions figure in [21].

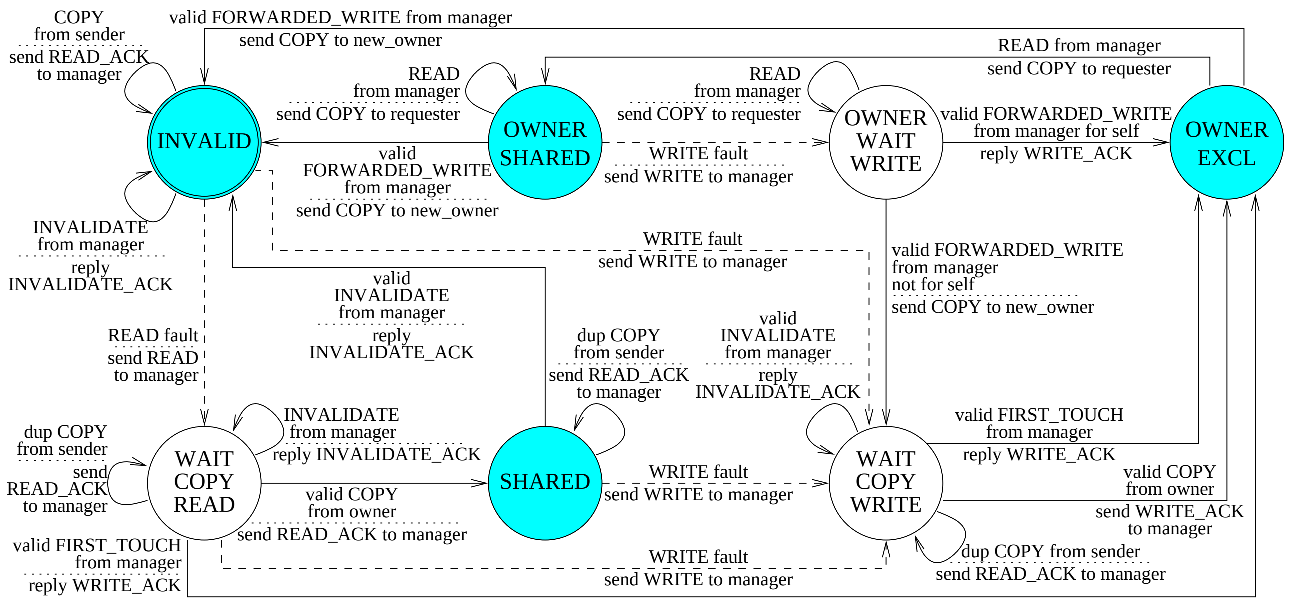

Figure 7: State Machine of a Node's Copy Supporting Messages Delivered in Disorder and Several Times

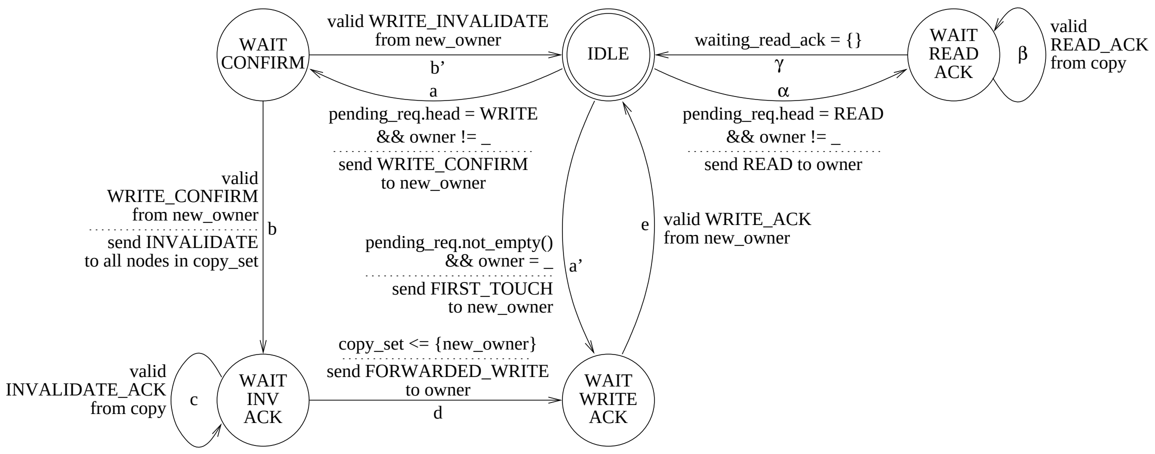
WAIT CONFIRM
IDLE
WAIT READ ACK
WAIT INV ACK
WAIT WRITE ACK
valid WRITE_INVALIDATE from new_owner
b'
a
pending_req.head = WRITE && owner != _
send WRITE_CONFIRM to new_owner
waiting_read_ack = {}
γ
α
pending_req.head = READ && owner != _
send READ to owner
β
valid READ_ACK from copy
valid WRITE_CONFIRM from new_owner
b
send INVALIDATE to all nodes in copy_set
pending_req.not_empty() && owner = _
a'
send FIRST_TOUCH to new_owner
e
valid WRITE_ACK from new_owner
copy_set <= {new_owner}
send FORWARDED_WRITE to owner
d
valid INVALIDATE_ACK from copy
c

Figure 8: Asynchronous Object Manager Supporting Messages Delivered in Disorder and Several Times

In some states and for some types of messages, the protocol distinguishes *dup* messages and *valid* messages. *dup* messages are detected as having already been delivered but nevertheless need replies to prevent the object manager from blocking. *valid* messages are detected as being delivered for the first time. If it is not specified, duplicated message deliveries are not detected.

To correctly detect *dup* messages, we assume that once a node has created a copy of the shared object, this node will not free the associated state related to the coherence protocol until either it disconnects or the application terminates.

The following paragraphs outline the differences with the FIFO protocol and discuss the gain in complexity.

### 4.2.1 Additions

To detect *dup* messages, the node's state machine uses two counters last_request and last_known_invalidate. The value of last_request is the number of READ and WRITE requests the node has sent since it has joined. This allows to detect *dup* COPY and FIRST-TOUCH messages. The value of last_known_invalidate is the number of WRITE requests that the OM has handled until the node has received its last copy. This allows to detect *dup* INVALIDATE messages that would cause the node to invalidate its copy when not needed, and *dup* forwarded WRITE messages that would cause the master copy to be sent to a node that has not requested it.

The OM's state machine maintains one counter last_invalidate that is the number of WRITE requests it has handled since its creation.

To prevent the master copy from being lost because of duplicated WRITE requests, the object manager's state machine includes the additional state WAIT_CONFIRM. In this state, the OM checks that the WRITE request it is handling is being handled for the first time. A more complete analysis figures in [21].

As a result, WRITE requests are handled atomically by following the (a, b, c, d, e) cycle of transitions if they are confirmed, or the (a, b') cycle otherwise. The first request can follow a shorter cycle (a', e) because the request can not have been handled before, and no valid copy exists.

### 4.2.2 Discussion

Compared to a FIFO communication channels-based asynchronous version of K. Li's protocol, our state machines are more complex. However, we discover that messages delivered in disorder need no adaptation: the acknowledgments, already needed in the FIFO case, enforce a sufficient order.

Surprisingly, coping with message delivered several times needs only a few counters. The gain over a generic message passing layer ensuring FIFO is clear: instead of maintaining a state for every other node, a node has to maintain only one counter summarizing a global state (last_known_invalidate), and a local state (last_request). Similarly, the object manager has to maintain only one counter summarizing a global state (last_invalidate).

Regarding performance, compared to the FIFO case, messages delivered in disorder induce no overhead. Concerning memory, message duplication has a very low cost: two counters per copy. Concerning execution speed, duplication makes the handling of WRITE requests slower, with 4 to 5 consecutive messages instead of 2 to 3, and can slow down the owner when it has write access, because its access rights are downgraded to read-only each time it receives a (duplicated) READ request.

## 5 Reconfiguration

In this section, we show the adjustments to the protocol presented in Section 4 to ensure liveness (that is the application never indefinitely blocks) despite up to $f$ simultaneous reconfiguration events.

### 5.1 Reconfigurations Affecting Nodes

#### Joins

Joins of nodes need no action with respect to nodes. Coherence is ensured because a joining node has no state. Liveness is also ensured because the DHT tolerates up to $f$ simultaneous reconfiguration events.

#### Disconnections

If nodes disconnect, the protocol presented above ensures coherence, but not liveness. Indeed, if the owner or a node in the copy set fails, the object manager may block when waiting for an acknowledgment. Similarly, if a node executing components of the application disconnects, the application may also block. Therefore the protocol uses component migration and injection of master copies in case of voluntary disconnections, and application checkpoint and restart in case of failures. We do not describe the checkpointing protocol in this paper, because this is orthogonal to the other mechanisms. Instead we rely on an existing protocol [5] that can be adapted with few changes to our system. To limit the cases needing restarts to failures of nodes executing components, we prune dependencies on nodes that do not execute components (see Table 1).

#### 5.1.1 Application Checkpoint and Restart

To tolerate failures of nodes executing components, we use application checkpoint and restart. To that end the application manager (AM) keeps track of the components' locations, and decides when the application should be restarted.

**Checkpoint** Although checkpointing and restart of distributed shared memory applications have been studied in several systems [18], efficient strategies are still a topic of research. In this paper, we assume that we dispose of a mechanism to compute coordinated checkpoints of a distributed shared memory application as described in [5], and do not focus at all on how to compute checkpoints. To summarize, a checkpoint includes the internal states of all the components and the content of the shared objects.

Checkpoints are stored and replicated in the system using fault-tolerant DHT storage such as PAST [23], which serves as system-wide stable storage. Each global checkpoint is assigned a unique ID. The AM always knows the ID of the last checkpoint, and hence can destroy previous useless checkpoints.

**Restart** The AM decides to restart the application when either it suspects (through a failure detector) a node executing a component to have disconnected, or it is asked to by an object manager. The object manager requests a restart when it suspects the owner or, during the handling of a WRITE request, the requesting node.

When restarting the application, the AM restores the components accordingly to their states in the checkpoint[3], and resets the object managers to state `IDLE` with an empty copy set and no owner.

The shared objects are lazily restored. Since the object managers have been restored with no owners, the first request of a node gets a FIRST-TOUCH message in reply. In this case, the node retrieves the value of the object from the checkpoint.

Since failure detectors are unreliable, components and copies may survive to a restart. In order to avoid inconsistencies, all messages passed between the entities of the application (object managers, nodes' copies, the AM, and components) include an epoch number. The epoch number is known by all the entities in a variable `epoch`, and is incremented by the AM before each restart. If an entity receives a message from a past epoch, the message is discarded. If the message comes from a future epoch, then the entity is a survivor and has to destroy itself.

**Tracking Components' Locations** To detect when a restart is needed, the AM must know at each time on which nodes the components execute. Information on components' locations changes when components are created, migrate, and terminate. By registering a component's location before it starts executing on a node, and by unregistering this location after the component has stopped executing on this node, the protocol ensures that at any time the AM monitors the nodes executing components, and some other nodes during short periods around component creation, migration, and termination.

### 5.1.2 Pruning Dependencies

We detail how dependencies on nodes executing no component are avoided. The resulting mechanisms allow to ensure that if a node is suspected to have failed, a restart is needed only if the node currently executes a component, or has ceased to execute components for the application a short time before and has not had enough time to inject all master copies of shared objects. In particular, right after a restart, since there are no valid copies of shared objects outside the checkpoints, there are no dependency to prune.

For a node $n$, actions to prune dependencies are taken in two cases: a) when node $n$ voluntarily disconnects, and b) when node $n$ ceases to execute components for the application (because the last component terminated or migrated). Both cases are handled in the same way. Indeed, since when node $n$ is in case a) all the components of the application migrate to other nodes, node $n$ is in case b) after the components have migrated.

A node that is not executing components does not keep copies of the application's shared objects because this is not beneficial. This results from the unreliability of failure detectors. Let us assume that a node $n$ can keep a copy even if it executes no component. If node $n$ disconnects, this copy becomes unreachable for the object manager (OM), and hence can not participate anymore to the coherence protocol. As a result, in order not to block and not to restart the application, the OM has to "forget" the copy. However, since failure detectors are unreliable, the OM may wrongly forget node $n$. As a result, the next component that executes on node $n$ must check by the OM that the copy is up to date. Such a checking may cost as much as directly requesting an up to date copy.

For this reason, when a node $n$ ceases to execute components for the application, node $n$ invalidates all its valid copies of shared objects. To that end, master copies have to be injected. These operations are detailed below for one shared object.

**Silent Invalidation of Cached Copies** If from the point of view of the OM node $n$ is not becoming the owner (that is node $n$ is in state `SHARED`, `WAIT_COPY_READ`, or `WAIT_COPY_WRITE` with no WRITE-CONFIRM message sent), node $n$ silently invalidates its copy without synchronizing with the OM. Avoiding to synchronize allows to use the same mechanisms for every type of disconnection.

When the OM suspects node $n$ because it blocks while waiting for a READ-ACK, INVALIDATE-ACK, WRITE-CONFIRM or WRITE-INVALIDATE message from node $n$, the OM asks the application manager

[3]The locations of the components can be chosen using help from the resource manager.

(AM) whether node $n$ is executing a component. The AM replies as soon as it knows that node $n$ is not executing any component. To simplify the algorithms, the reply of the AM is the message that the OM is waiting from node $n$ for. Indeed, if the AM replies, then node $n$ should have invalidated its copy, and can be safely removed from the copy set. Otherwise, the AM eventually decides a restart if node $n$ is executing a component and has disconnected. Hence, the OM can not block indefinitely.

**Injection of Master Copies** If node $n$ is owner, or is becoming owner from the point of view of the OM (that is node $n$ is in state WAIT_COPY_WRITE with a WRITE-CONFIRM message sent), a new owner must be chosen. To that end, node $n$ routes an INJECTION request to the AM. If the last component leaving the node has migrated, the request can include a hint indicating that a good new owner might be the target node of the migration. When receiving an INJECTION request, the AM chooses a node executing at least one component[4], and sends to this node a BECOME-OWNER request. A node receiving a BECOME-OWNER request simulates a WRITE fault, and hence eventually becomes the owner.

The injection does not complete if node $n$ or the new owner disconnects before. In these cases, the application is restarted thanks to the mechanisms described in paragraph 5.1.1. As a result, when node $n$ disconnects, a restart is needed only if node $n$ has just ceased to execute components *and* the injection of a master copy has not completed.

## 5.2 Reconfigurations Affecting Managers

### 5.2.1 Replicated State Machine

Fault tolerance and reachability for object managers and application managers (more simply called managers thereafter) is achieved using active replication of the state machines [25] and their failure detectors. We actively replicate object managers in order to allow them to be spread on a large number of nodes, depending on the mechanism used to generated their IDs in the DHT, without lowering the mean time between failures of entities of the application.

Since failure detectors may be not deterministic, they must be logically separated from the state machines of the managers. As a result, the state machines activate and deactivate their failure detectors, and the failure detectors generate inputs for their associated state machines.

The determinism of the state machines allows them to be replicated using atomic multicast [17] for their inputs. Such replication schemes can be made efficient for little numbers of replicas [7]. With failure detectors in class $\diamond S$, ensuring liveness despite up to $f$ disconnections requires $2f + 1$ replicas.

### 5.2.2 Mapping Replicas to Nodes

Mapping replicas to nodes involves two issues. First, the system must be able to choose $2f + 1$ distinct nodes so that replicas are always reachable *via* the DHT. To that end, we use the same scheme as in fault-tolerant DHT storage [23]. The replicas are mapped to the $2f + 1$ nodes closest to the key in the ID space. This needs that the logical neighborhood of nodes includes at least $2f + 1$ nodes.

Second, consequently to disconnections or joins, some nodes may have to be added to or removed from the replica group. In order to preserve the coherence of the replicas, the changes to the group are done using a Primary Component Group Membership service [10]. In order to detect when the group membership should be changed, each member of the group monitors its logical neighborhood. When a member detects that a change is needed, it requests the other members to change the group membership. The change is accepted if a quorum of members agrees to. This should prevent from changing the group membership too frequently because of temporary inconsistencies in the nodes' neighbor sets during reconfigurations. The architecture proposed in [17] for dynamic group communication systems provides the required flexibility to implement the desired service.

[4]The policy used to choose the new owner should be implemented by the resource manager, and is thus beyond the scope of this paper.

## 5.3 Proof of Correctness

We prove two properties of the protocol assuming one shared object: coherence (safety) and liveness. Our definition of coherence is the following: according to the global real time clock[5] , the real times of the write accesses to the shared object are totally ordered, and each read access returns the value written at the last preceding time of a write access (with respect to the global clock).

Liveness means that once a node requests a copy of the object it eventually receives a reply, unless the application is restarted afterwards.

We prove the preceding properties assuming that all the events that can occur figure in the model we have described in Section 3. Moreover, to prove the liveness property, we also assume that no more than $f$ reconfiguration events occur per reconfiguration window.

In this paper, we present the main parts of the proofs. The interested reader can find complete proofs in [21]. The proofs that we present here have been simplified for better understanding. We outline the proofs in paragraph 5.3.1. The impatient reader can skip the following paragraphs that detail the proofs and directly read Section 6.

### 5.3.1 Outline of the Proofs

The proof for the coherence property relies on the causality that the object manager (OM) can deduce from the reception of valid acknowledgments (thanks to the detection of duplicated messages). If the message comes from a node $n$, then this causality expresses that node $n$ is no more involved in any action of the protocol, unless it has started another one, which the OM has not started to handle. If the message comes from the application manager (AM) for a node $n$, then node $n$ executed no components, has no valid copy, and has at most been able to start a request, which the OM has not started to handle.

The proof for the liveness property consists of two parts. From the point of view of a node, we show that every request is eventually satisfied, unless the application is restarted afterwards. From the point of view of the OM, we show that when the OM blocks while waiting for a message that has been lost, the AM either unblocks the OM or restarts the application.

### 5.3.2 Valid Messages

In the proofs we give in this paper, we assume that *dup* messages and *valid* messages are correctly detected by the protocol. We also extend the validity of messages as follows.

DEFINITION 1 At time $t$, a message $m$ of the protocol being transported towards an entity $e$ is *valid* if, assuming that entity $e$ delivers message $m$ at time $t$, entity $e$ considers message $m$ as valid (that is not *dup*).

In the following paragraphs, $M$ references the OM.

### 5.3.3 Coherence

The combination of the three following properties, lemma 1, and lemma 3 show that 1) each valid copy has the same value as the master copy, and 2) the master copy is transferred from owner to owner in the same order as the real times of their write accesses. As a result, the protocol ensures coherence.

PROPERTY 1 A write access from a component to the local copy of the object succeeds only when the copy is in state OWNER_EXCLUSIVE.

PROPERTY 2 If a node sends a *valid* COPY message, then its copy is in state OWNER_EXCLUSIVE, or OWNER_SHARED, or OWNER_WAIT_WRITE.

[5]Remember that no node has access to this clock

PROPERTY 3 The copy of a node $n$ becomes valid only when node $n$ receives a *valid* COPY or FIRST-TOUCH message.

LEMMA 1 If a *valid* COPY message is being transported towards a node at time $t$, then, at time $t$, we have $M.\mathsf{state} \in \{\mathsf{WAIT\_READ\_ACK}, \mathsf{WAIT\_WRITE\_ACK}\}$.

LEMMA 2 (COPY SET) If a node $n$ has a valid copy and node $n$ is not owner, then we have $n \in M.\mathsf{copy\_set}$.

*Proof:* Before a node $n$ gets a valid copy, it must send a READ request to manager $M$. Manager $M$ adds node $n$ to $M.\mathsf{copy\_set}$ before forwarding the request to the owner, and hence before node $n$ gets a valid copy.

Node $n$ is removed from $M.\mathsf{copy\_set}$ when manager $M$ is in state $\mathsf{WAIT\_INV\_ACK}$ and receives a valid INVALIDATE-ACK message for node $n$ at time $t_{\mathrm{ack}}$ (transition c). Let $t_{\mathrm{inv}}$ be the time at which manager $M$ executed transition b and entered state $\mathsf{WAIT\_INV\_ACK}$. The INVALIDATE-ACK message comes a) either from node $n$, b) or from the AM to unblock manager $M$.

In case a), since the INVALIDATE-ACK message is valid, the copy of node $n$ has been invalid since at least some time $t$, with $t_{\mathrm{inv}} < t < t_{\mathrm{ack}}$. As a result, according to lemma 1, the copy of node $n$ remains invalid at least until the next time $t_{\mathrm{n}}$, with $t_{\mathrm{n}} > t_{\mathrm{ack}}$, at which manager $M$ enters state $\mathsf{WAIT\_READ\_ACK}$ or $\mathsf{WAIT\_WRITE\_ACK}$.

In case b), since manager $M$ asked the AM whether node $n$ executes a component after $t_{\mathrm{inv}}$, there is a time $t_{\mathrm{no\ component}}$, with $t_{\mathrm{inv}} < t_{\mathrm{no\ component}} < t_{\mathrm{ack}}$, at which node $n$ executes no component. Since at time $t_{\mathrm{inv}}$ the copy of node $n$ was neither a master copy, nor a copy in state $\mathsf{WAIT\_COPY\_WRITE}$ with a WRITE-CONFIRM message sent (otherwise no invalidation of the copy of node $n$ would have been requested), there has been no component on node $n$ at time $t_{\mathrm{no\ component}}$, and hence the copy of node $n$ has been invalid since at least some time $t$, with $t_{\mathrm{inv}} < t < t_{\mathrm{ack}}$ (see paragraph 5.1.2). We can conclude as in case a) with the same notations. □

LEMMA 3 (EXCLUSIVE OWNER) If a node $o$ has a copy in state $\mathsf{OWNER\_EXCLUSIVE}$, then (i) no other node has a valid copy, and (ii) no valid COPY or FIRST-TOUCH message is being transported towards any node.

*Proof:* In order to show the result, we examine what happens between the times a node enters and exits state $\mathsf{OWNER\_EXCLUSIVE}$.

Before node $o$ enters state $\mathsf{OWNER\_EXCLUSIVE}$ at time $t_{\mathrm{o}}$, it must have sent at time $t_{\mathrm{r}}$ (with $t_{\mathrm{r}} < t_{\mathrm{o}}$) a WRITE request to manager $M$. Manager $M$ ensures that all copies (lemma 2) but the copy of node $o$ and the copy of the owner $o_{\mathrm{o}}$ are invalid before it forwards the WRITE request to the owner and enters state $\mathsf{WAIT\_WRITE\_ACK}$ (transition d) at time $t_{\mathrm{f}}$ (with $t_{\mathrm{r}} < t_{\mathrm{f}} < t_{\mathrm{o}}$). Let $t_{\mathrm{ack}}$, $t_{\mathrm{o}} < t_{\mathrm{ack}}$, be the time at which manager $M$ exits state $\mathsf{WAIT\_WRITE\_ACK}$. If $o \neq o_{\mathrm{o}}$, when node $o_{\mathrm{o}}$ sends a COPY message for the WRITE request of node $o$ at time $t_{\mathrm{s}}$ (with $t_{\mathrm{f}} < t_{\mathrm{s}} < t_{\mathrm{o}}$), it enters state $\mathsf{INVALID}$. As a result, since after time $t_{\mathrm{f}}$ manager $M$ does neither enter state $\mathsf{WAIT\_READ\_ACK}$ nor transits from state $\mathsf{WAIT\_INV\_ACK}$ to state $\mathsf{WAIT\_WRITE\_ACK}$ before time $t_{\mathrm{ack}}$, lemma 1 shows that, between times $t_{\mathrm{o}}$ and $t_{\mathrm{ack}}$, only node $o$ has a valid copy and no valid COPY message is being transported towards any node.

A node can also enter state $\mathsf{OWNER\_EXCLUSIVE}$ if it receives a FIRST-TOUCH message. The same arguments as above apply because 1) *dup* FIRST-TOUCH messages are detected, 2) when manager $M$ sends a FIRST-TOUCH message, no node has a valid copy of the object, and 3) when manager $M$ sends a FIRST-TOUCH message, it enters state $\mathsf{WAIT\_WRITE\_ACK}$ (transition a').

As soon as a node having a copy in state $\mathsf{OWNER\_EXCLUSIVE}$ sends a COPY message, this node enters state $\mathsf{OWNER\_SHARED}$ or state $\mathsf{INVALID}$. As a result, assertion (ii) and hence assertion (i) remain valid. □

### 5.3.4 Liveness

It is noticeable that if messages are lost, coherence is not endangered. The proof for liveness however relies on the assumption that if a message is sent, then it is eventually received unless the target node or the sender crashes. In particular, the liveness relies on the fact that no manager crashes as long as no more than $f$ reconfiguration events occur per reconfiguration window. First we show that the protocol ensures liveness when no disconnection occurs, and second we examine the cases in which disconnections can endanger liveness.

Under the assumption that no node disconnects, lemmas 4 and 5 show that when a node sends a request to manager $M$, this request is eventually satisfied.

LEMMA 4 If $M.\mathsf{state} \neq \mathsf{WAIT\_WRITE\_ACK}$, then node $M.\mathsf{owner}$ is the owner. If manager $M$ sends a FORWARDED-WRITE request to node $M.\mathsf{owner}$ (transition d), then node $M.\mathsf{owner}$ is the owner, and remains owner until it receives the request.

In particular, the master copy is never lost.
*Proof:* Manager $M$ forwards WRITE requests only if they are confirmed. □

LEMMA 5 If a node $n$ sends a request to manager $M$, then node $n$ eventually receives a *valid* COPY message or a *valid* FORWARDED-WRITE message.

*Proof:* Since no message is lost, manager $M$ eventually receives the request. Since no message is lost and, according to lemma 4, all requests are forwarded to the true owner, every request that manager $M$ removes from the queue eventually receives a reply. Since the queue is a FIFO, node $n$ eventually receives a reply. □

Now we show that, despite disconnections, manager $M$ never indefinitely blocks, and that if it gets unblocked thanks to a message from the AM replacing a message from a node $n$, then node $n$ does not block either. This suffices because a) manager $M$ never crashes and b) every request from a node $n$ is sent to manager $M$, and terminates by an acknowledgment that manager $M$ waits for.

Manager $M$ may indefinitely block when it waits for a message from a node $n$ in state $\mathsf{WAIT\_INV\_ACK}$, or $\mathsf{WAIT\_CONFIRM}$, or $\mathsf{WAIT\_WRITE\_ACK}$, or $\mathsf{WAIT\_READ\_ACK}$, and node $n$ disconnects before the message is delivered.

We examine the four cases. In state $\mathsf{WAIT\_INV\_ACK}$, as soon as its failure detector suspects node $n$, manager $M$ asks the AM to reply when node $n$ executes no component. If the AM never replies, then node $n$ executes a component. In this case the AM eventually decides a restart (completeness of the failure detectors).

In state $\mathsf{WAIT\_CONFIRM}$, as soon as manager $M$ suspects node $n$, it asks the AM to restart the application. Since failure detectors are complete, this eventually happens.

In states $\mathsf{WAIT\_WRITE\_ACK}$ and $\mathsf{WAIT\_READ\_ACK}$, manager $M$ blocks either because node $M.\mathsf{owner}$ disconnects before sending the COPY message to node $n$, or because node $n$ disconnects before sending the acknowledgment. In state $\mathsf{WAIT\_WRITE\_ACK}$, as soon it suspects node $n$ or node $M.\mathsf{owner}$, manager $M$ asks the AM to restart the application. Hence, liveness is ensured.

In state $\mathsf{WAIT\_READ\_ACK}$, as soon as it suspects node $n$, manager $M$ asks the AM to reply when node $n$ executes no component. As soon as it suspects node $M.\mathsf{owner}$, manager $M$ asks the AM to restart the application. Since failure detectors are complete, this ensures the liveness of manager $M$.

A node $n$ does not block if manager $M$ is unblocked by the AM because the only messages replaced are acknowledgments terminating the requests.

## 6 Experimental Evaluation

We implemented our protocol using a DHT based on the algorithms of Pastry [22]. We evaluated our protocol in “real-world” executions and by simulation.

To show the practical interest of our protocol, we measured the speed-up obtained using our data sharing service in a parallelized application.

To observe the impact of reconfiguration, we simulated the execution of a simple 1 producer/$n$ consumers application and compared the execution speeds both in a static configuration and in a highly dynamic configuration.

We have not implemented state machine replication yet. As a result, we ensured that the application managers and the object managers ran on stable nodes.

## 6.1 Setup for "Real-World" Executions

We executed the MGS application. MGS produces an orthonormal basis from an independent set of vectors, using the Modified Gram-Schmidt algorithm. For each iteration, a component computes a new vector of the basis, and all components use this vector to correct the remaining vectors to standardize. The distribution of vectors used is cyclic. The shared objects are the vectors.

We executed MGS with basis of $512$, $1024$, and $2048$ vectors. For each size, we executed MGS on $1$, $3$, $5$, $7$, and $10$ nodes. Each node executed 1 component. For all numbers of nodes, the components synchronized using a barrier, and a separate node served as barrier manager.

The nodes are HyperThreaded Intel Pentium IV Xeon bi-processors running at 2.4GHz, having 1 GByte of RAM, and interconnected by a Fast Ethernet switch.

## 6.2 Setup for Simulations

We coupled our prototype to a simple discrete-event simulator. The simulator provides a message-passing module that replaces the TCP module used in real-world runs. The message-passing module simulates a latency of 100ms between each pair of nodes. The simulator supports trace-based injection of reconfiguration events, and provides an abstraction of component.

### 6.2.1 Reconfiguration Traces

To model a dynamic system, we used traces of reconfiguration events derived from real-world measurements in the Gnutella peer-to-peer system.

The Gnutella trace gives the arrival and departure times of about $17000$ nodes monitored during $60$ hours and probed every seven minutes [24]. In our simulations, we used $1000$ nodes that were chosen randomly in the traces.

### 6.2.2 Application Modeling

A component is modeled as a sequence of atoms. We used mainly 3 types of atom: compute locally during a specified amount of time, read or write a shared object, enter a barrier.

On top of this component abstraction we implemented simple application checkpoint and restart operations. We used a global coordinated checkpointing strategy. These operations are instantaneous in simulation time. In particular, checkpointing has no overhead.

When a node fails, and if the application should then be restarted, the application is immediately restarted. This models the effect of perfect failure detectors.

### 6.2.3 Applications Simulated

We simulated two versions of a 1 producer/$n$ consumers application. The first version is synchronized and the second one not. In the synchronized version (SPC), each component iterates on a loop composed of two phases. In the first phase, the producer computes a value and writes it to the shared object. In the second phase, the consumers read the value from the shared object, and do a treatment. Each phase ends in a barrier. This application models for example an experiment of high energy physics in which a measuring instrument produces data that need a parallel processing.

The computation times of the producer and the consumers both equal to 1s. The application executed $1000$ loops. The application was checkpointed before each loop.

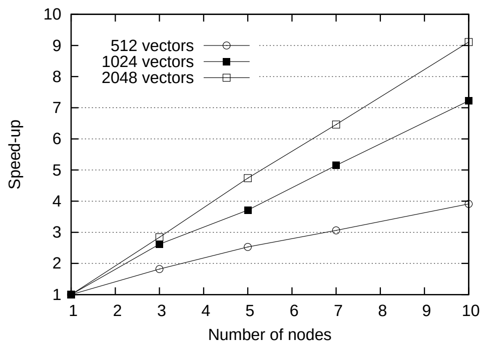

Figure 9: The speed-up obtained on MGS becomes linear for great numbers of vectors

In the unsynchronized version (UPC), each component iterates on a loop composed of a computation and an access to the shared data. The first component writes to the shared data. The other components read the shared data. The components only synchronize before the first loop and after the last loop in a barrier. This application models for example the loose coupling of two codes of simulation iterating at different steps. The second code uses the output of the first code as initial conditions at each simulation step.

To further study the impact of concurrency, we modified UPC first to execute 10 concurrent producers (10 producers/$n$ consumers), and second to execute as many concurrent producers as consumers ($n$ producers/$n$ consumers).

The computation times were randomly generated and exponentially distributed with a mean of 1s. The applications executed $1000$ loops.

In a first experiment we executed SPC in a static configuration of $500$ nodes. We made $n$ vary from 1 to $499$. For each value of $n$, the components were located on consecutive nodes in the overlay ring.

In a second experiment we executed UPC in a static configuration of $2000$ nodes. We made $n$ vary from $10$ to $1000$. For each value of $n$, the components were located on consecutive nodes in the overlay ring.

In a third experiment we executed SPC in a dynamic configuration of $1000$ nodes. We made $n$ vary from $1$ to $200$. The average number of nodes connected to the system during the simulations was $600$. We chose one stable node of ID $0$ on which the application manager and the object manager were mapped by choosing their respective keys equal to $0$. In order to compare with the static configuration, we made the application start as soon as at least $n + 2$ nodes (including the stable node) were connected to the system. At starting time as at each restart, we mapped the components to separate nodes using a simple round robin algorithm.

For each type of configuration, we distributed the node IDs regularly and uniformly in the ID space.

## 6.3 Experimental Results

### 6.3.1 Speed-Up of MGS

We measured the speed-up of the MGS application with the "real-world" prototype on 1 to $10$ nodes.

The larger the size of the basis is, the better the speed-up is (Figure 9). Indeed, the ratio between computations and communications increases with the number of vectors. Moreover, since each vector is in a separate shared object, no conflict between read and write accesses occur.

The reasonable speed-ups obtained show that our protocol is of practical interest.

### 6.3.2 Synchronized 1 Producer/$n$ Consumers Application in a Simulated Static Confi guration

We measured the execution times of the synchronized $1$ producer/$n$ consumers application (SPC) in the static configuration. We made the measurement start after the initializations of the application manager and the object manager.

| Number of consumers | $\in [1, 11]$ | $\in [12, 499]$ |
|---|---|---|
| Total execution time (s) | 2899.6 | 3099.5 |
| Execution time (s) per iteration, the first one excepted | 2.9 | 3.1 |
| Execution time (s) of the first iteration | 2.5 | 2.6 |

Table 2: Execution times of 1000 loops of SPC

We obtained 2 execution times depending on the number of components (Table 2). Up to 12 components, the execution time is $2899.6$s, and for more than 12 components the execution time is $3099.5$s. This results from the routing algorithms of Pastry. Indeed, up to 12 components, node $0$, on which the object manager is located, figures in the neighborhood of all components, whereas for more than 12 components, some consumer components are at $2$ routing hops of node $0$. This makes INVALIDATE-ACK messages and READ requests reach the object manager in 200ms, which induces an overhead of 200ms per iteration. Since no copy exists before the first iteration, no invalidation is requested, and the overhead is only 100ms.

We can also compare these execution times with a message passing version of SPC. Using the same model as in the simulator, we would obtain execution times of 2100s if the producer pushes the data to the consumers, and 2200s if the consumers pull the data from the producer. This assumes however an optimal message passing interface. Current approaches for fault-tolerant message passing in large scale systems like MPICH-V would produce times of at least 2200s and 2400s respectively because of the indirection of messages through checkpoint servers [6]. Since our coherence protocol implements a pull scheme, our data sharing service produces an overhead of 20% to 30%. To obtain more precise results however, we should take into account the atomic multicast mechanism that is used by both the state machine replication scheme of the object managers of our protocol and the checkpoint servers of MPICH-V.

In these results, we observe that thanks to efficient routing algorithms in the overlay, using a distributed hash table to locate managers impacts lowly on performance. Moreover, compared to fault-tolerant message passing-based communications, our data sharing service makes a reasonable trade-off between performance and ease of use.

### 6.3.3 Unsynchronized 1 Producer/$n$ Consumers Application in a Simulated Static Configuration

In order to study the effect of contention, we measured the times needed to obtain access rights in the unsynchronized 1 producer/$n$ consumers application (UPC) executed in a static configuration (Figure 10).

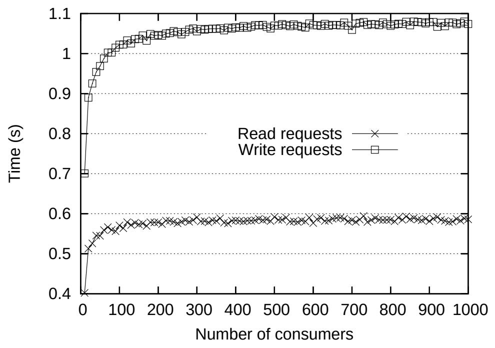

Figure 10: Mean Times to Obtain Access Rights in UPC

Compared to SPC, access rights are obtained in longer times. For write accesses the time is about 1.1s (compared to 0.7s in SPC), and for read accesses the time is about 0.6s (compared to 0.4s in SPC). In both cases, this is an increase of about 50%. This increase results from the concurrency of read and write accesses. Interestingly, these times are near constant as the number of consumers increases. This suggests that our protocol handles well the scheme of contention produced by one writer and many readers.

To study the impact of concurrent write accesses, we also executed a 10 producers/$n$ consumers version and a $n$ producers/$n$ consumers version of UPC.

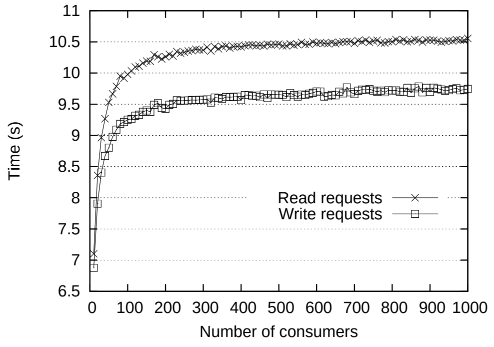

Figure 11: Mean Times to Obtain Access Rights in Unsynchronized 10 Producers/$n$ Consumers

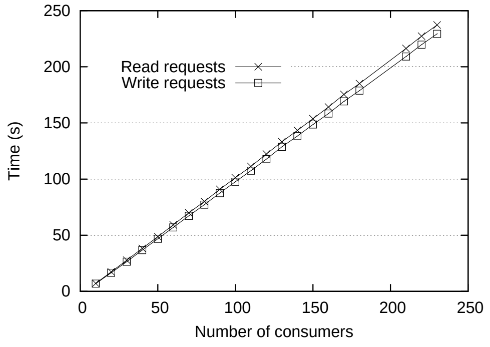

Figure 12: Mean Times to Obtain Access Rights in Unsynchronized $n$ Producers/$n$ Consumers

The times to obtain access rights are much longer when several components write concurrently. This confirms the intuition that allowing concurrent writers greatly increase computation times. However, when the number of producers is low in comparison to the number of readers, the access times still are near constant. This confirms that our protocol scales well with the number of readers.

In the extreme case in which the number of producers is the same as the number of consumers, the access times increase linearly with the number of components.

For write access times higher than a few seconds, the time indicates the average number of write requests that are queued in the object manager's request queue. Indeed, since all requests for read access queued between two consecutive write requests in the request queue are handled in parallel, and since a write request terminates in at most 0.8s once the object manager picks it from the queue, the average delay for a write request to be served is between $0.8 \times l$ and $1.4 \times l$ seconds, where $l$ is the average length of the request queue at the time at which a write request is inserted.

As a result, we observe that with high numbers of producers, on average all consumers are waiting for a write access, and all consumers are waiting for a read access.

These results show that data sharing is scalable with respect to the number of consumers, even when read and write accesses occur concurrently. These results also show that the times to obtain access rights depend mostly on the number of concurrent writers, and seem to increase linearly with the number of concurrent writers. These observations may change when taking bandwidth into account, since in the simulations infinite bandwidth are assumed and hence READ requests are truly handled in parallel.

### 6.3.4 Synchronized 1 Producer/$n$ Consumers Application in a Simulated Dynamic Configuration

We studied the impact of reconfiguration on the execution of the synchronized 1 producer/$n$ consumers application. Like in the static configuration, we started to measure the execution times once the application manager and the object manager were initialized.

In the dynamic configuration, the applications were restarted several times because of failures of nodes executing components. The resulting execution times are less than three times the execution time obtained in the static configuration, and even less than two times in most cases (Figure 13 and Table 2).

The starting times of the applications are not regular (Figure 13). This results from the initial placements of the components, that were not chosen the same for different numbers of consumers. As a result, some

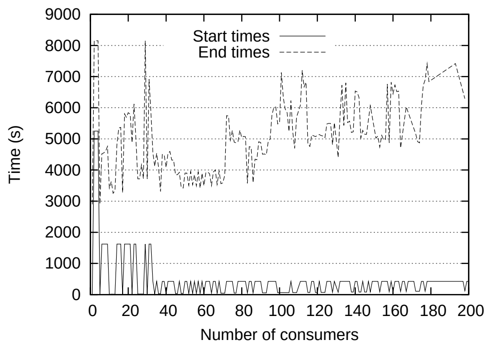

Figure 13: The number of components increases the slow-down induced by reconfigurations.

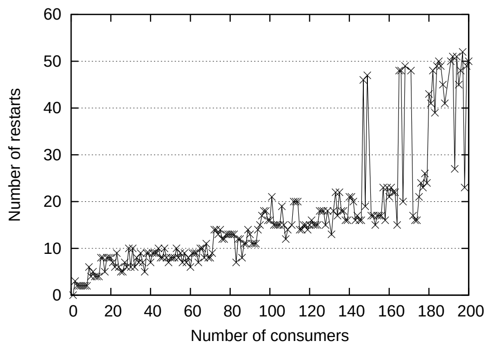

Figure 14: Number of Restarts

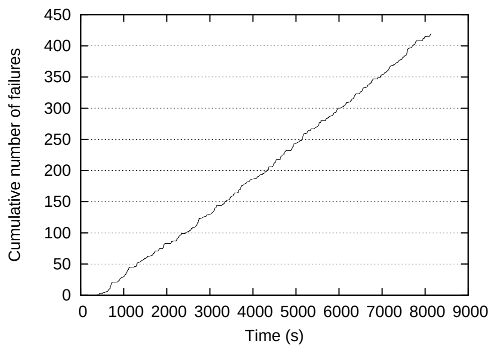

Figure 15: Cumulative Number of Node Failures

components were placed on nodes at long routing distances from the object manager's node. These long routing distances resulted from the imperfection of the routing tables caused by the frequent reconfiguration events (Figure 15). The late start times result from long initialization times of the object manager.

The number of restarts increases with the number of components (Figure 14). However, compared with the cumulative number of failures (Figure 15) since the starting times of the applications, the number of restarts is approximately 1/10 of the number of failures, even when the application spans up to 1/3 of the connected nodes.

The execution times of the application increase with the number of components. The more components the application has, the more frequently the application fails. But we also observed that some nodes had so imperfect routing tables that it was impossible for components on these nodes to route messages to the object manager. In such cases, the application was unblocked at the next restart, when a new placement was chosen (using the round robin policy). Most of these cases however should disappear thanks to a better implementation of Pastry's algorithms.

This experiment has shown that reconfiguration has a high impact on the execution times. However, the impact results more from the imperfection of the routing tables of the overlay, which slows down communications, than from restarts of the application. Moreover, the traces we used were derived from a highly dynamic environment of a peer-to-peer file sharing application used on the Internet. In comparison with Gnutella, Grids are much less dynamic. As a result, we can expect much better execution times in a less dynamic environment like a computational Grid.

## 7 Conclusion and Future Work

We have presented a transparent data sharing service for distributed applications in a dynamic distributed system like a grid. Our service is the first, to our knowledge, that ensures atomic consistency of cached shared data and the liveness of applications sharing data on an asynchronous distributed system despite simultaneous reconfiguration events including failures.

Our consistency protocol is based on the write-invalidate scheme to provide strong guarantees on consistency despite reconfiguration. The data sharing service tolerates every reconfiguration event in the system's life time, provided that no more than a fixed number of reconfiguration events occur simultaneously. To ensure the liveness of the application, the protocol replicates state machines and uses backward error recovery if components of the application fail. The protocol uses a distributed hash table provided by an overlay network to transparently locate the managers and choose the replicas of their state machines.

Such a protocol provides a reliable basis to build a data sharing service in the Grid that truly alleviates the burden of the user and the programmer to manage data location, caching, and consistency.

We have implemented the data sharing service using a Pastry-like [22] overlay network. The experimental evaluation shows that despite an additional complexity introduced by a model of dynamic distributed system, our protocol allows to obtain reasonable speed-ups. Compared to fault-tolerant message passing-based communications, our data sharing service provides a reasonable trade-off between performance and ease of use. The protocol scales well with the number of readers, even if a writer concurrently accesses the same data. Moreover, we have noticed on simulations that the overhead that dynamic joins and leaves of nodes induce on execution speeds results more from the increased routing delays in the overlay network than from the restarts the applications undergo. Although these observations need to be confirmed with more realistic applications, this suggests that the performance obtained should be better on Grids, which are less dynamic environments than peer-to-peer Internet file-sharing applications.

We are currently studying variants of the protocol to improve performance. In particular, to avoid using atomic multicast of the requests for access rights, we are relaxing the synchronism between the replicas of object managers. To further lower the latency of requests for access rights, we are also adapting the consistency protocol to remove the object managers from the main path of requests.

## References

[1] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. *Kluwer Journal of Supercomputing*, 2004. To appear. Available as INRIA Report RR-5082.

[2] Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM consistency protocol hierarchy-aware: an efficient synchronization scheme. In *Proc. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 516–523, Tokyo, May 2003. Held in conjunction with CCGRID 2003.

[3] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. In *Proc. ACM Workshop on Adaptive Grid Middleware (AGRIDM 2004)*, Antibes Juan-les-Pins, France, September 2004. To appear.

[4] Luciana Arantes, Pierre Sens, and Bertil Folliot. An effective logical cache for a clustered LRC-based DSM system. *Cluster Computing*, 5(1):19–31, January 2002.

[5] Ramamurthy Badrinath, Christine Morin, and Geoffroy Vallée. Checkpointing and recovery of shared memory parallel applications in a cluster. In *Proceedings of the International Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 471–477, Tokyo, May 2003. Held in conjunction with CCGrid 2003.

[6] G. Bosilca, A. Bouteiller, F. Cappello, S Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *IEEE Proc. of SuperComputing*, 2002.

[7] Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, January 2001. Technical Report MIT/LCS/TR-817.

[8] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[9] DeQing Chen, Chunqiang Tang, Brandon Sanders, Sandhya Dwarkadas, and Michael L. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proceedings of the ninth symposium on Principles and Practice of Parallel Programming*, pages 131–142, June 2003.

[10] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[11] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *Peer-to-Peer Systems II (IPTPS 2003)*, pages 33–44, February 2003.

[12] Pascal Gallard, Christine Morin, and Renaud Lottiaux. Dynamic resource management in a cluster for high-availability. In *Euro-Par 2002: Parallel Processing*, volume LNCS 2400, pages 588–592, Paderborn, Germany, August 2002. Springer-Verlag.

[13] Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of The 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–286, October 1998.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 272–281, June 1997.

[16] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Peer-to-Peer Systems II (IPTPS 2003)*, pages 21–32, February 2003.

[17] Sergio Mena, André Schiper, and Pawel Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of International Middleware Conference*, pages 414–432. Springer, June 2003.

[18] Christine Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(9), September 1997.

[19] Frank Mueller. On the design and implementation of DSM-Threads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 315–324, June 1997.

[20] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*. Usenix, March 2003.

[21] Louis Rilling and Christine Morin. A coherence protocol for cached copies of volatile objects in peer-to-peer systems. Research Report RR-5059, INRIA, December 2003.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350, 2001.

[23] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP'01)*, pages 188–201, 2001.

[24] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN) 2002*, San Jose, CA, USA, January 2002.

[25] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[26] Hazim Shafi, Evan Speight, and John K. Bennett. Raptor: Integrating checkpoints and thread migration for cluster management. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 141–152. IEEE, CS Press, October 2003.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399