HAL
archives-ouvertes.fr

A Coherence Protocol for Cached Copies of Volatile Objects in Peer-to-Peer Systems

Louis Rilling, Christine Morin

▶ To cite this version:

Louis Rilling, Christine Morin. A Coherence Protocol for Cached Copies of Volatile Objects in Peer-to-Peer Systems. [Research Report] RR-5059, INRIA. 2003. inria-00071524

HAL Id: inria-00071524
https://hal.inria.fr/inria-00071524

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Coherence Protocol for Cached Copies of Volatile Objects in Peer-to-Peer Systems

Louis Rilling, Christine Morin

N°5059

Décembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5059--FR+ENG

INRIA
RENNES

A Coherence Protocol for Cached Copies of Volatile Objects in Peer-to-Peer Systems

Louis Rilling*, Christine Morin*

Thème 1 — Réseaux et systèmes
Projet Paris

Rapport de recherche n°5059 — Décembre 2003 — 25 pages

Abstract: We consider the problem of executing distributed applications using the shared memory paradigm on dynamic and large scale distributed systems, such as structured peer-to-peer systems. The shared memory is private to an application, volatile, and components of the application transparently access shared memory objects via their usual address space. The peer-to-peer system tolerates up to f simultaneous reconfiguration events (node failure, disconnection, or join) and an infinite number of reconfigurations. We give a coherence protocol similar to K. Li's protocols for cached copies of memory objects. The protocol uses the peer-to-peer architecture to handle up to f simultaneous reconfiguration events and an infinite number of reconfigurations with a fail-stop/recovery model. Failures are tolerated using backward error recovery and replicated automata to avoid restarts of applications when possible. We prove that the protocol ensures coherence of cached copies of memory objects despite reconfiguration events, and ensures liveness if communications are reliable.

Key-words: Fault-Tolerant Distributed Shared Memory, Self-Organizing Distributed Systems, Structured Peer-to-Peer Systems, Checkpointing, Replicated Automata

(Résumé : tsvp)

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

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un protocole de cohérence séquentielle pour des copies en cache d'objets volatiles dans les systèmes pair-à-pair

Résumé : Nous considérons le problème consistant à exécuter des applications distribuées utilisant le paradigme de la mémoire partagée sur des systèmes distribués dynamiques et de grande échelle, tels les systèmes pair-à-pair structurés. La mémoire partagée n'est accessible que par une application, elle est volatile, et les composants de l'application y accèdent de manière transparente via leur espace d'adressage usuel. Le système pair-à-pair tolère jusqu'à f événements de reconfiguration (défaillance d'un nœud, déconnection, ou connection) simultanément, et une infinité de reconfigurations. Nous donnons un protocole de cohérence séquentielle similaire aux protocoles de K. Li pour des copies en cache d'objets en mémoire. Le protocole utilise l'architecture pair-à-pair pour gérer jusqu'à f événements de reconfiguration simultanément, et une infinité de reconfigurations, en prenant un modèle défaillance silencieuses avec recouvrements. Les défaillances sont tolérées en utilisant des recouvrements en arrière et des automates dupliqués pour éviter au possible de redémarrer les applications. Nous prouvons que le protocole assure la cohérence séquentielle des copies en cache d'objets en mémoire malgré l'occurrence d'événements de reconfiguration, et assure la vivacité si les communication sont fiables.

Mots-clé : mémoire distribuée virtuellement partagée tolérante aux fautes, systèmes distribués auto-gérés, systèmes pair-à-pair structurés, points de reprise, automates dupliqués

1 Introduction

We consider the problem of executing distributed applications using the shared memory paradigm in dynamic and large scale distributed systems, such as structured peer-to-peer systems. Peer-to-Peer (P2P) systems are naturally designed with the main concerns of scalability, dynamicity, and hence tolerance to multiple simultaneous failures. Such systems have mostly been studied for shared file storage, and recent works provide file systems ensuring from NFS-like to sequential consistency of data[15]. 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 dynamic behaviour of the nodes, that is unexpected simultaneous connections, disconnections and failures (referenced as reconfiguration events thereafter)[18, 6, 15].

By contrast, we are interested in ensuring sequential consistency (more simply named coherence in the paper) for volatile shared data accessed through volatile memory. Two main differences with file sharing are noticeable. First, in order to have acceptable execution speeds, such accesses have to be performed as much as possible on the local memory of the node. Therefore we have to ensure coherence of cached copies of shared objects. Second, shared objects are private to an application. In particular, shared objects are not persistent. Hence coherence of the objects has only to be ensured with respect to the application, which allows to use backward error recovery.

While the execution of shared memory applications on fault-tolerant distributed systems has been well studied, those systems were mainly static clusters of workstations with at most a few hundreds of nodes[12]. In particular, the system tolerate only one node failure, and must reconfigure itself before it can tolerate an other failure[20, 8]. Moreover, only one node can be removed or added per reconfiguration. By contrast, peer-to-peer systems can cover geographically large areas, and hence comprise large numbers of nodes that join or leave the system at will. Thus, the rate of failure is increased, and it is important to tolerate several simultaneous reconfiguration events.

We present a coherence protocol for cached copies of volatile objects in a structured peer-to-peer system. The protocol tolerates up to f simultaneous reconfiguration events per reconfiguration if the underlying peer-to-peer system tolerates at least f reconfiguration events per reconfiguration. We describe the system in Section 2, and then present the protocol in Sections 3 and 4. In Section 3 we assume that some components of the protocol, namely object managers and application managers, are always reachable, and show how to tolerate any number of simultaneous reconfiguration events under these assumptions. In Section 4 we remove these assumptions and show how to tolerate f simultaneous reconfiguration events. Finally, Section 5 concludes the paper.

2 Assumptions

The main contribution of our protocol is that it works in a peer-to-peer (P2P) environment. In this section, we describe more precisely this environment.

2.1 Peer-to-Peer System

A P2P network has mainly two properties: dynamicity (nodes may join, leave, or fail at any time) and symmetry (no *a priori* distinction of capacity or functional capability between nodes). Our protocol assumes that up to f reconfiguration events may occur simultaneously, *i.e.* in a same time window, called reconfiguration window, that corresponds to the time necessary to reconfigure the system after a reconfiguration event.

More precisely, our protocol targets structured P2P networks[17]. Although several such P2P systems have been proposed, they have common properties and can be used through a common interface[7]. Therefore we describe the main properties we assume on the P2P system, without presenting in details one system. The system's nodes are fully connected, using IP routing for example. Each node is

assigned a unique identifier (ID) on s bits (typically $s = 128$, 160 or 256). Interesting properties, like load balancing and independent node failures, can be deduced if the IDs are randomly generated (for example using a secure hash function, like SHA-1, applied on the IP address) and uniformly distributed in the ID space.

Basically, the P2P system provides a distributed hash table (DHT), by dynamically mapping keys to nodes. To that end, a key is represented by an identifier on s bits, and is mapped to the closest node in the ID space. In order to keep this mapping correct despite reconfiguration events, each node keeps up to date a local set of its neighbours in the ID space, called *neighbour set*. When a node sends a message to a key, the message is forwarded from node to node using the nodes' neighbour sets, until it reaches the node responsible for the key. As a result, the system is organized in an overlay network, in which each node maintains a routing table based on its neighbour set.

2.2 Disconnections, Failures, and Asynchrony

Although structured P2P systems deal with byzantine failures, which include arbitrary link failures (message loss, corruption, etc.), our coherence protocol assumes a fail-stop/recovery model: nodes can fail silently, and recover later on. Indeed, tolerating byzantine failures need that threads of distributed shared memory applications be replicated, which causes a serious slow-down.

We handle link failures that cause temporary and persistent disconnections. A node disconnects when it gets unreachable, and reconnects when it gets reachable again. A node disconnects either voluntary, either due to a link failure, either because it fails. If it is not specified otherwise, disconnections enclose the three cases. In a fail-stop model, a node may not distinguish between link failures and failures of other nodes. For this reason, disconnections due to link failures and node failures are both referenced as node failures in the paper. Consequently, the protocol ensures that the fail-stop assumption remains valid despite reconnections.

The system is asynchronous: there is no bound on the time needed for a message to reach its destination. Moreover, we assume that nodes dispose of failure detectors allowing to detect if a specified node is unreachable. Failure detectors achieve strong completeness: if a node n disconnects, there is a time after which the failure detector *suspects* node n. We also assume that the nodes in a specified group of nodes dispose of failure detectors in class $\diamond S$[4], which achieve strong completeness and eventual weak accuracy: there is a time after which some reachable node in the group is never suspected by any node in the group.

2.3 Message Passing

Entities that communicate are identified by keys, grouped in services. Entities communicate using two levels of messages. The primitive $\mathsf{send}(n, k, s, m)$ sends message m to key k in service s on node n using the native network (IP for example). The primitive $\mathsf{route}(k, s, m)$ sends message m to key k in service s using the P2P network. The primitive $\mathsf{route_replica}(r, k, s, m)$ is similar to route, but additionally multicasts the message to the r nodes closest to the key in the ID space. The primitive $(k, s, m) = \mathsf{recv}()$ receives a message, the sender's key and service and the message being returned in k, s, and m respectively.

The structured P2P API described in [7] assumes messages routed through the P2P overlay may be lost, be corrupted, be delivered out of order, and even duplicate. In order to simplify the protocol's description, we assume that there is no message loss nor corruption. Solutions to prevent message loss have been studied in [11, 2]. With a fail-stop model for node failures, message corruption can only be generated by the network. This problem has been extensively dealt with in the literature and therefore we will not discuss it any further.

3 Coherence Protocol

We designed our coherence protocol for volatile shared objects. Typically, such objects are mapped in the address space of threads of an application, and we assume that shared objects are only accessed by threads of one application. In this section, we assume without loss of generality that there is only one shared object.

The protocol is similar to K. Li's invalidation-based coherence protocol using fixed distributed managers[10]. Mainly two types of entities participate to the protocol: the nodes, which can have copies of the shared object (more simply referenced as *copies* thereafter), and an object manager, which handles nodes' requests for copies so that coherence is preserved. Assigning copies to nodes instead of threads allows threads to migrate transparently with respect to the coherence protocol. Nodes are represented by their node IDs, and the object manager is represented by a key in the P2P system. There is always exactly one *master copy.* The node having the master copy is called the *owner.* A node becomes owner when it gets a writable copy. Additionally an application manager is used to ensure fault tolerance for threads. The application manager is also represented by a key in the P2P system.

Three separated services implement the three entities. To send a message to a node, entities use primitive send. To send a message to a manager, entities use primitive route.

Our protocol improves previous works based on K. Li's algorithms by tolerating up to f simultaneous reconfiguration events per reconfiguration window, and allowing any number of reconfigurations in the system's life time. Dynamicity is enabled using the DHT behaviour of the underlying P2P system. Fault tolerance is reached using checkpointing for threads and shared objects, and replication, like in DHTs[18, 6], for the object managers and the application manager. We also try to minimize the number of cases in which the application has to be restarted.

In order to simplify the protocol's description, we assume in this section that the object manager and the application manager never disconnect. We ensure that this hypothesis is verified in Section 4.

3.1 Basic Coherence Protocol

First we describe the protocol with the additional assumption that no node disconnects. We remove these assumptions in Section 3.2. Joining of new nodes does not matter, because nodes' and managers' IDs do not change, and the underlying P2P DHT ensures correct forwarding of messages despite joins.

Our protocol is very similar to K Li's protocol based on fixed distributed managers. Therefore we recall quickly this protocol before we show the contributions of our protocol.

We choose to handle duplication and lack of order of messages at the protocol level instead of the message passing layer, because 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, in some cases due to the replication of the object manager (see Section 4), duplicated messages may be desirable and hence must be handled at the protocol level.

3.1.1 K. Li's Protocol Based on Fixed Distributed Managers

Node's Site 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 the access is read-only, 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 following properties ensure coherence: when the owner is in state OWNER_EXCLUSIVE, all other copies are in state INVALID; when the access rights for a copy are not sufficient, the node retrieves a copy with suitable access rights from the owner.

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 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.

At the beginning of the execution, no node has a copy. Therefore 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.

Manager's Site 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.

3.1.2 Asynchronous State Machines and Out of Order Message Delivery

The behaviours of nodes and object managers are described using deterministic input/output state machines. The point using deterministic input/output state machines is explained in Section 4.

Similarly to the protocol implemented in DSM-Threads[13], both state machines are designed to work asynchronously: message receptions occur only before transitions. Besides the benefits in flexibility, this is required for active replication (see Section 4). As a consequence, the node's state machine includes more states that are wait points for transitions between the main states. Figures 1 and 2 show the state machines respectively for the object manager (OM) and for a node. For readability purposes, only the conditions to apply the transitions are shown in the figures. The actions taken at each transition figure in Appendix A. The following paragraphs outline the features of the algorithms.

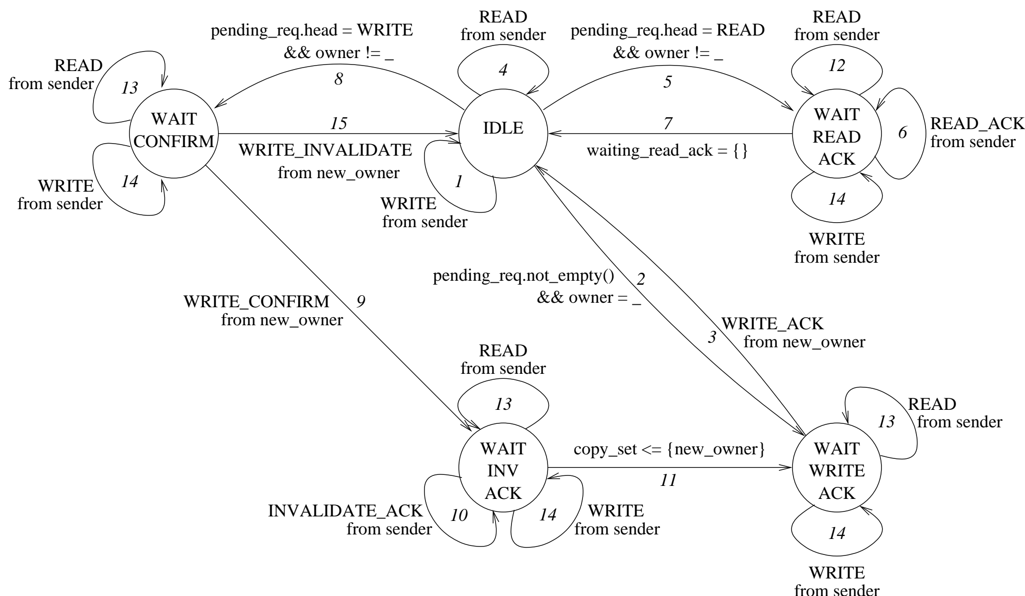

Figure 1: Automaton of the Object Manager

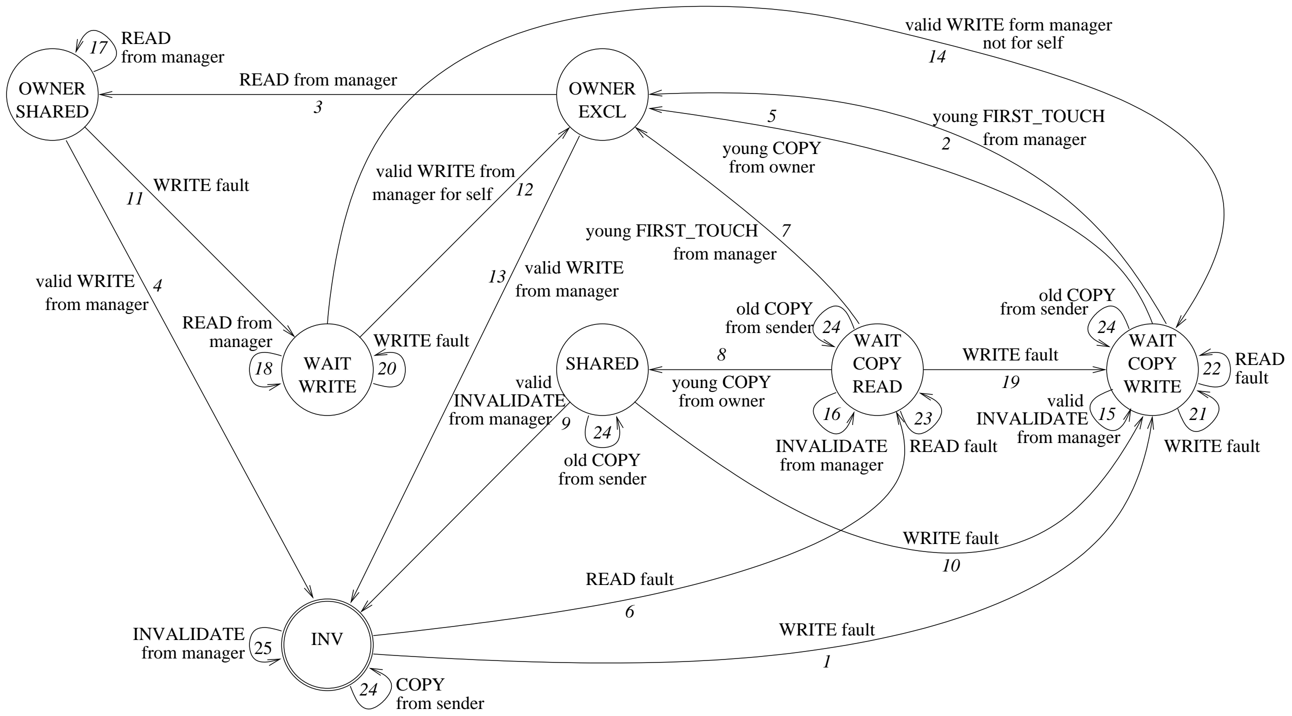

Figure 2: Automaton of a Node's Copy

The OM queues node's requests in pending_req and handles them using a FIFO (First In First Out) 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.

Since messages may be delivered out of order, we add acknowledgements (READ_ACK messages and WRITE_ACK messages) to handle WRITE requests atomically, which ensures coherence of the copies and liveness of the system. Otherwise a WRITE request from the owner followed by a READ request at the manager's site might arrive after the READ request at the owner's site, resulting in a state in which, simultaneously, the owner can write to the object and an other node can read the object. We can also build scenarios in which requests are lost because the owner changes between the time the requests are redirected and the time they arrive.

In 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) in transitions 5 and 12 but not acknowledged yet. As a result, WRITE requests and all subsequent requests are queued until all previous requests are terminated.

In state WAIT_INV_ACK and state WAIT_WRITE_ACK, a WRITE request from node new_owner is being processed. As a result, all subsequent requests are queued until the WRITE request is terminated. The OM requests all the nodes in copy_set but node new_owner to invalidate their copies in transition 9, and waits in state WAIT_INV_ACK for their acknowledgements. The OM removes a node from copy_set in transition 10 when the node acknowledges its invalidation. Once all invalidations are acknowledged, the OM redirects the WRITE requests to node owner in transition 11, and waits for the acknowledgement from node new_owner in state WAIT_WRITE_ACK. The OM receives the acknowledgement from the new owner and commits the change of owner (owner := new_owner) in transition 3.

3.1.3 Handling of Duplicated Messages

In order to allow the protocol to rely on causality[9] between sending and delivery of some messages, we ensure, despite messages that duplicate, that some messages are delivered *exactly once*, and, for replies to requests, only if they are *young*, *i.e.* replies to pending requests. The protocol requires this for COPY, FIRST-TOUCH, and INVALIDATE-ACK messages to ensure coherence, for READ-ACK and WRITE-ACK messages to ensure coherence and liveness (see paragraph 3.1.2), and for WRITE messages to ensure liveness (otherwise, the master copy may get lost when the owner transfers it to a node that does not want it).

For all the messages cited above but WRITE requests, exactly once and young delivery is achieved using per entity sequence numbers for READ, WRITE, and INVALIDATE requests.

Exactly once and young delivery for READ-ACK, COPY, FIRST-TOUCH, and INVALIDATE-ACK messages is achieved, for READ requests and WRITE requests, using per node sequence numbers, named last_req, and for INVALIDATE requests, a sequence number maintained by the object manager, named lats_inv. Each messages sent consecutively to a READ, WRITE, or INVALIDATE request, includes the appropriate sequence number. The message is considered as young if the sequence number included matches the local sequence number.

Ensuring exactly once delivery for WRITE requests from any node n is harder because node n and the object manager have no *a priori* knowledge about the state of each other. To solve the problem, in transition 8, the manager asks node n to confirm its request with a WRITE-CONFIRM message that includes the sequence numbers of the WRITE request and the sequence number last_inv, incremented before the object manager sends the WRITE-CONFIRM message. Node n either confirms the request by sending back the WRITE-CONFIRM message, or invalidates the request by sending a WRITE-INVALIDATE message including the same sequence numbers.

Node n confirms the request if it is still in state WAIT_COPY_WRITE or in state WAIT_WRITE, and its counter last_req value equals the value included in the WRITE-CONFIRM message.

WRITE requests forwarded to the owner also need exactly once delivery. We do this using a different message type, named FORWARDED-WRITE, that includes the original WRITE request and the value of the object manager's counter last_inv. The owner keeps this value in a local variable last_known_inv and updates it when it receives FORWARDED-WRITE requests including higher values. This value is included in the COPY messages, so that the owner always knows the sequence number of the last invalidation. As a result, exactly once and young delivery can be ensured for WRITE-ACK messages if they include the sequence number last_known_inv of the new owner.

Receiving a duplicated READ request causes at worst a COPY message to be sent to a node that will not take it into account because the COPY message received will not be young, and the copy set to include nodes that have no valid copy. Such nodes should however reply with READ-ACK and INVALIDATE-ACK messages in order to prevent the object manager from blocking. Liveness could be endangered because in the worst case, a duplicated READ request makes the owner having just entered state OWNER_EXCLUSIVE enter state OWNER_SHARED, without having time to write to the object. However, using FORWARDED-READ requests including the sequence number of the last invalidation can lower the rate of such duplicated messages. An other solution is to delay the treatment of the request for a little time, sufficient to ensure that the execution progresses. Such solutions are known as *freezing*.

Duplicated INVALIDATE requests may cause at worst invalidations of copies, which does not affect coherence. This does not affect liveness for similar reasons to the case of duplicated READ requests satisfied by the owner when it is in state OWNER_EXCLUSIVE. However, this may affect performance greatly.

Fortunately, we can ensure exactly once delivery of INVALIDATE requests for nodes having valid copies with no additional cost, using the sequence number last_known_inv in the node's state. When a node receives a valid COPY message, it stores the included sequence number of the last preceding

invalidation in its variable last_known_inv. Nodes consider INVALIDATE requests valid only if their sequence number is greater than the the value of last_known_inv.

Since the owner always knows the sequence number of the last invalidation, and this number is included in the COPY messages, and after an invalidation phase no node but the owner has a valid copy of the message, a node having a valid copy always knows the sequence number of the last invalidation phase. As a result, old invalidations are not considered valid. If a node has no valid copy and receives an INVALIDATE request, it can safely apply the same algorithm. In the worst case, useless duplicated INVALIDATE-ACK messages will be sent, against which the protocol is already protected.

3.2 Disconnections of Nodes Having Copies

If nodes having copies disconnect, the protocol presented above still preserves 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 acknowledgement. Therefore the protocol needs some adjustments that use the application manager.

In many aspects, voluntary disconnections and failures are not distinguishable, and both events generate reconfigurations of the system. However, voluntary disconnections allow to prepare reconfigurations and make them less costly. First we present a basic strategy using backward error recovery, and second we show how to optimize the strategy.

3.2.1 Basic Strategy: Application Checkpoint and Restart

Our basic strategy to tolerate disconnections and failures consists in restarting the application each time a node, having a valid copy or a thread of the application, disconnects. To that end the application manager keeps track of the threads' locations, and decides when the application should be restarted.

Checkpoint Although checkpointing and restart of distributed shared memory applications has been studied in several systems[12], efficient strategies are still a topic of research. In this paper, we assume we dispose of a mechanism to compute coordinated checkpoints of a distributed shared memory application as described in [1]. As a result, a checkpoint includes the internal states of all the threads and the content of the shared address space. If disconnections occur during a checkpoint, at worst the checkpoint is aborted.

Checkpoints are stored and replicated in the system using fault-tolerant DHT storage such as PAST[18] or CFS[6]. Each global checkpoint is assigned a unique ID and the application manager always knows the ID of the last checkpoint. After having ensured that a new checkpoint has been saved with enough replication, the application manager can destroy the previous checkpoint.

Restart The application manager decides to restart the application when either it detects that a node executing a thread has disconnected, either it is asked to by an object manager. Since the cause may be a link failure or a node failure, between which the managers can not distinguish, threads and copies may survive to a restart.

In order to avoid conflicts between the restarted application and surviving parts of the previous execution, all messages exchanged between the entities of the application (object managers, nodes' copies, the application manager, and threads) include an epoch number that is known by all the entities in a variable epoch, and is incremented before each restart. If an entity receives a message from a different epoch, the message is discarded. Moreover, if the message comes from a future epoch, then the entity is a survivor and has to destroy itself.

To restart the application, the application manager, named *am* thereafter, executes the following steps:

1. increment counter *am*.epoch;

2. destroy the reachable threads that are still running and all the reachable copies of shared objects;
3. reinitialize the object managers to state IDLE with an empty copy set, no owner, and the value of am.epoch as epoch number;
4. use the last available global checkpoint to recreate the threads in a frozen state, the epoch number set to the value of am.epoch;
5. unfreeze the threads.

The shared address space is lazily restored. After a restart, since the object managers have been restored with no owners, the first request of a node gets a FIRST-TOUCH message in reply. In that case, the node executes a thread, which knows the ID of the checkpoint used to restart. Hence, the node retrieves the value of the object from the checkpoint.

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

Tracking Threads' Locations In order to detect if a thread becomes unreachable, the application manager must know at each time on which nodes the threads are located. Informations on threads' locations change when threads are created, migrate, and terminate.

The following rules ensure that at each time, every thread is monitored by the application manager. When a thread is created, the application manager records its location before the thread begins its execution. Before a thread is migrated, the application manager records the new location of the thread, and forgets the previous location only once the migration has completed. When a thread terminates, the application manager is notified only once the thread has stopped executing.

Additionally, the threads' locations are completely reset when the application is restarted.

Disconnection Detection Disconnections are detected using failure detectors. The application manager monitors the nodes executing threads, and the object manager monitors the owner and the nodes in the copy set.

The application manager is implemented by a deterministic asynchronous input/output state machine for the same reasons as object managers are (see Section 4). However, failure detectors may be not deterministic, and hence must be logically separated from the state machines' core. As a result, the state machines activate and deactivate their failure detectors, and the failure detectors generate inputs for the state machines.

3.2.2 Avoiding Restarts

Restarting the application has an obvious cost. Therefore we optimize voluntary disconnections, and restrict cases that need restarts to failures of nodes executing threads and failures of owners.

Thread Migration Before a node n voluntarily disconnects, it migrates the threads it is executing to other nodes[1]. However, if node n has also a copy of the shared object, the restart is not avoided yet. As a result, we also have to optimize voluntary disconnections of nodes having *alone* copies, that is without threads.

Early Invalidation of Alone Copies Disconnections of nodes without threads are handled using early invalidation of the shared object's copies. When a node n having an alone copy disconnects, this copy becomes unreachable for the object manager, and hence can not participate anymore to the coherence protocol. As a result, in order not to block, the object manager has to "forget" the

[1] The target nodes can be chosen using help from the resource manager.

Case	Voluntary Disconnection	Failure
Invalid copy and no thread	-	-
Shared copy and no thread	invalidate	invalidate
Master copy and no thread	inject	lazy restart
Invalid copy and threads	migrate	restart
Shared copy and threads	invalidate + migrate	restart
Master copy and threads	inject + migrate	restart

Table 1: Cases and actions to handle voluntary disconnections and failures of nodes

copy. However, in case of a temporary link failure, the object manager may decide that node n has disconnected, while node n may not detect the failure. As a result, the next thread that executes on node n must check by the object manager 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 the last thread executing on a node n leaves the node (because it terminated or migrated), node n invalidates all its valid copies of shared objects.

Silent Invalidation of Shared Copies A node that has a copy in state SHARED, state WAIT_COPY_READ, or state WAIT_COPY_WRITE with no WRITE-CONFIRM message sent, silently puts its copy in state INVALID, without synchronization with the manager. This allows to use the same mechanisms for every type of disconnection.

When the object manager suspects a node n in the copy set, and may block because it is waiting for a READ-ACK or INVALIDATE-ACK message from node n, it asks the application manager whether node n executes a thread. The application manager replies as soon as it knows that node n executes no thread. To simplify the algorithms, the application manager's reply is the message that the object manager is waiting from node n for. Indeed, if the application manager replies, then node n should have invalidated its copy, and can be safely removed from the copy set. Otherwise, either the disconnection was temporary, and the object manager must wait until the node becomes reconnects, either a thread of the application has been lost in node n, and the application manager will eventually decide a restart (completeness property of the failure detectors).

Injection of Master Copies For master copies, and copies in state WAIT_COPY_WRITE with a WRITE-CONFIRM message sent, a new owner must be chosen. To that end, the node issues an INJECTION request to the application manager. If the last thread leaving the node has migrated, the request can include a hint indicating that a good new owner could be the target node of the migration. When receiving an INJECTION request, the application manager chooses a node executing at least one thread[2], and sends it a BECOME-OWNER request. A node receiving a BECOME-OWNER request simulates a WRITE fault, and hence eventually becomes owner.

If the injection fails, because it can not complete before the node disconnects or the new owner fails, the application is restarted, using the same mechanisms as described in paragraph 3.2.1. Once the application has restarted, there are no valid copies of shared objects outside the checkpoints. Hence, at most one restart due to a loss of owner may occur per node disconnection, which would be the default behaviour if disconnections were not optimized.

Summary Table 1 shows a summary of the cases we identified and the corresponding actions. The cases for a node result from the state of the shared object's copy and the presence of threads of the application.

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

3.3 Proof of Correctness

We prove two properties of the protocol: coherence and liveness. Coherence is roughly defined by the rule "the value read is the last value written". Liveness means that once a node requests a copy of the object, it receives a reply within a finite delay, unless the application is restarted afterwards.

The first result concerns exactly once and young deliveries (see paragraph 3.1.3).

LEMMA 1 The rules mentioned in Section 3.1.3 are correct: messages identified to need exactly once (and young) delivery are delivered and considered valid exactly once (and only if they are young).

Proof: The proof relies on the fact that sequence numbers of READ, WRITE, WRITE-CONFIRM, and INVALIDATE requests are initialized to 0, and incremented before a new request is sent. In particular, all requests have sequence numbers greater than 0. Sequence numbers of READ and WRITE requests from a node n are unique among all the READ and WRITE requests from node n, and sequence numbers of WRITE-CONFIRM (respectively INVALIDATE) requests are unique among all the WRITE-CONFIRM (respectively INVALIDATE) requests.

COPY messages A node n considers a COPY(v,s,i) message m as young only if node n is in state WAIT_COPY_READ or WAIT_COPY_WRITE and n.last_req $= s$. Since message m can only have been sent by the owner consecutively to a reception of a READ(n,s) or FORWARDED-WRITE(n,s,i) message, and sequence numbers of READ and WRITE requests are unique, message m is young, and will not be considered as young a second time.

WRITE-CONFIRM, WRITE-INVALIDATE, and INVALIDATE-ACK messages Using the object manager's last_inv counter, similar arguments can be used for WRITE-CONFIRM, WRITE-INVALIDATE, and INVALIDATE-ACK messages received by the object manager. Moreover, since the object manager does not change its value last_inv after having sent a WRITE-CONFIRM or INVALIDATE request until it receives a young reply, replies are considered as young at least once, and hence exactly once.

WRITE-ACK messages A WRITE-ACK(n,s) message m is considered as valid by the object manager only if it is in state WAIT_WRITE_ACK, $n =$ new_owner and $s =$ last_inv. Since sequence numbers of INVALIDATE requests are unique, message m is young, and will not be considered as valid a second time. Moreover, since the object manager does not change the value of last_inv between the time it forwards a WRITE request to the owner and the time it waits for a WRITE-ACK message, WRITE-ACK messages are considered as valid at least once.

FORWARDED-WRITE messages The owner considers a FORWARDED-WRITE(n,ws,is) request as valid only if $is >$ last_known_inv. Since the owner updates its counter last_known_inv in this case, at most once delivery is ensured for FORWARDED-WRITE requests. Since exactly once and young delivery is ensured for WRITE-ACK messages, and since the object manager waits for a WRITE-ACK message, consecutive to a FORWARDED-WRITE request, before it sends an other FORWARDED-WRITE request to the owner, it is ensured that the value last_known_inv of the owner is lower than the value included in a FORWARDED-WRITE request that have been sent but not received. Hence, FORWARDED-WRITE requests are delivered at least once. As a result, exactly once delivery is ensured for FORWARDED-WRITE requests.

READ messages Since READ messages can be delivered more than once, READ-ACK messages can also be delivered more than once. However, it is ensured that COPY messages are considered as young at most once, and the first time they are received. As a result, no young COPY message can

be received after the first READ-ACK message that is considered as young is received. As a result, READ-ACK messages that are considered as young can be considered as young, because this does not endanger the causality between the reception of a valid COPY message and the reception of a valid READ-ACK message.

FIRST-TOUCH messages The remainder of the proof shows that exactly once and young delivery is ensured for FIRST-TOUCH messages. A FIRST-TOUCH message m can be considered valid by node n only if (i) node n is in state WAIT_COPY_READ and n.last_req $= 1$, or (ii) node n is in state WAIT_COPY_WRITE and n.last_req $\in \{1, 2\}$ and n.last_known_inv $= 0$, that is node n has never had a copy before, and is waiting for a reply. Indeed, since the object manager has sent a FIRST-TOUCH message, node n is the first node for which the object manager has treated a request for a copy. Conditions (i) and (ii) ensure that a FIRST-TOUCH message is considered as young at least once, because node n can not send a READ request of sequence number greater than 1 without receiving a reply to the previous request, and can not send a WRITE request of sequence number greater than 2 without receiving a reply to a previous request. Indeed, only transition 19 allows to send a request without having received a reply to the previous request.

Now we show that FIRST-TOUCH messages are considered as young at most once, which concludes the proof.

Condition (i) is verified at most once, because of the unicity of sequence numbers. Condition (ii) is verified with n.last_req $= 1$ at most once, in which case condition (i) can never be verified when receiving a FIRST-TOUCH message. Hence, to conclude the proof, we have to show that condition (ii) is verified with n.last_req $= 2$ at most once when receiving a FIRST-TOUCH message, and that in that case, condition (i) can never be verified when receiving a FIRST-TOUCH message.

If node n is in state WAIT_COPY_WRITE and n.last_req $= 2$, then a) either node n has executed transition 19, that is before receiving the reply to the READ request numbered 1, in which case message m can be safely considered as young because transition 2 and 7 are identical, b) either node n has executed transition 1. Node n can not have executed transition 10 because, to enter state SHARED, node n should have received a young COPY message in reply to its READ request numbered 1, and hence not have been the first node for which the object manager has treated a request for a copy. For similar reasons, node n can not have executed transition 14.

In case b), since n.last_req $= 2$, node n has entered state INVALID from an other state, before executing transition 1. Since all transitions that enter state INVALID exit states in which node n has a valid copy, node n has invalidated its copy at least once, either by transferring the ownership in reply to a WRITE request, either in reply to an INVALIDATE request. As a result, in case b), we have n.last_known_inv $\neq 0$. Hence if condition (ii) is true, this results from case a).

When receiving a FIRST-TOUCH message, case a) is verified at most once thanks to the unicity of sequence numbers, and in that case condition (i) can never be verified when receiving a FIRST-TOUCH message. As a result, condition (ii) is verified at most once when receiving a FIRST-TOUCH message, in which case condition (i) can never be verified when receiving a FIRST-TOUCH message.

Consequently, when receiving a FIRST-TOUCH message, only one of condition (i) and (ii) can be verified, in which case neither condition (i) nor condition (ii) can be verified when receiving a subsequent FIRST-TOUCH message. □

Now we extend the validity of messages, as defined in paragraph 3.1.3, to messages being transported.

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.

In the following paragraphs, the object manager will be simply referenced by M.

3.3.1 Coherence

The combination of the three following properties, lemma 2, and lemma 4 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. As a result, the protocol ensures coherence.

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

PROPERTY 2 Only a node having a copy in state OWNER_EXCLUSIVE, state OWNER_SHARED, or state WAIT_WRITE can send a *valid* COPY message.

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

LEMMA 2 If a *valid* COPY message is being transported towards a node at time t, then, at time t, the object manager is in state WAIT_READ_ACK or in state WAIT_WRITE_ACK.

Proof: Manager M enters state WAIT_READ_ACK before forwarding any READ request to the owner. Moreover, manager M exits state WAIT_READ_ACK only if all READ requests that have been forwarded to the owner have been acknowledged by the requesters with READ-ACK messages, for which young delivery is ensured according to lemma 1. As a result, no valid COPY messages replying to READ requests can be sent by the owner if manager M is not in state WAIT_READ_ACK.

The arguments are similar with state WAIT_WRITE_ACK, entered by sending FORWARDED-WRITE requests, and exited by receiving WRITE-ACK messages, for which exactly once and young delivery is ensured. □

LEMMA 3 (COPY SET) If a node n has a valid copy and node n is not owner, then $n \in M$.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.copy_set before forwarding the request to the owner (transitions 5 and 12), and hence before node n gets a valid copy.

Node n is removed from M.copy_set when manager M is in state WAIT_INV_ACK and receives a valid INVALIDATE-ACK message for node n at time t_{ack} (transition 10). Let t_{inv} be the time at which manager M executed transition 9 and entered state WAIT_INV_ACK. The INVALIDATE-ACK message comes a) either from node n, b) either from the application manager to indicate that node n has no thread.

In case a), since the INVALIDATE-ACK message is young, the copy of node n has been invalid since at least some time t, with $t_{\text{inv}} < t < t_{\text{ack}}$. As a result, according to lemma 2, the copy of node n remains invalid at least until the next time t_{n}, with $t_{\text{n}} > t_{\text{ack}}$, at which manager M enters state WAIT_READ_ACK or WAIT_WRITE_ACK.

In case b), since manager M asked the application manager whether node n executes a thread after t_{inv}, there is a time $t_{\text{no thread}}$, with $t_{\text{inv}} < t_{\text{no thread}} < t_{\text{ack}}$, at which node n executes no thread. As a consequence, since at time t_{inv} the copy of node n was not a master copy, nor a copy in state WAIT_COPY_WRITE with a WRITE-CONFIRM message sent (otherwise no invalidation of the copy of node n would have been requested), the copy of node n has been an alone copy at time $t_{\text{no thread}}$, and hence has been invalid since at least some time t, with $t_{\text{inv}} < t < t_{\text{ack}}$ (see paragraph 3.2.2). We can conclude as in case a) with the same notations. □

LEMMA 4 (EXCLUSIVE OWNER) If a node o has a copy in state OWNER_EXCLUSIVE, then (i) no other node has a valid copy, and (ii) no valid COPY 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 OWNER_EXCLUSIVE.

Before node o enters state OWNER_EXCLUSIVE at time t_{o}, it must have sent at time t_{r} (with $t_{\mathrm{r}} < t_{\mathrm{o}}$) a WRITE request to manager M, which ensures that all copies (lemma 3) but the copy of node o and the copy of the owner o_{o} are invalid before forwarding the request to the owner and entering state WAIT_WRITE_ACK (transition 11) at time t_{f} (with $t_{\mathrm{r}} < t_{\mathrm{f}} < t_{\mathrm{o}}$). Let t_{ack}, $t_{\mathrm{o}} < t_{\mathrm{ack}}$, be the time at which manager M exits state WAIT_WRITE_ACK. If $o \neq o_{\mathrm{o}}$, when node o_{o} sends a COPY message for the WRITE request of node o at time t_{s} (with $t_{\mathrm{f}} < t_{\mathrm{s}} < t_{\mathrm{o}}$), it enters state INVALID (transition 4, 13, or 14). As a result, since manager M does not enter state WAIT_READ_ACK nor transits from state WAIT_INV_ACK to state WAIT_WRITE_ACK before time t_{ack}, lemma 2 shows that, between times t_{o} and t_{ack}, only node o has a valid copy of the object and no valid COPY message is being transported to any node.

A node can also enter state OWNER_EXCLUSIVE if it receives a FIRST-TOUCH message. The same arguments as above apply because 1) exactly once and young delivery is ensured for FIRST-TOUCH messages, 2) when manager M sends a FIRST-TOUCH message, no node has a valid copy of the object, and 3) manager M enters state WAIT_WRITE_ACK by sending a FIRST-TOUCH message (transition 2).

As soon as a node having a copy in state OWNER_EXCLUSIVE sends a COPY message, it enters state OWNER_SHARED (transition 3) or state INVALID (transition 13). As a result, assertion (ii) and hence assertion (i) remain valid. □

3.3.2 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. 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 every node is always reachable, lemmas 5 and 6 show that when a node sends a request to manager M, it receives a reply within a finite delay.

LEMMA 5 If manager M is not in state WAIT_WRITE_ACK, then node M.owner is the owner. If manager M sends a FORWARDED-WRITE request to node M.owner (transition 11), then node M.owner is the owner, and remains owner until it receives the request.

In particular, the master copy is never lost.

Proof: The first node that becomes owner in an epoch is the node that receives a valid FIRST-TOUCH message (transitions 2 and 7). At that time, manager M is in state WAIT_WRITE_ACK. A new node becomes owner when it receives a valid COPY in reply to a WRITE-REQUEST (transition 5). At the time a node becomes owner, manager M is in state WAIT_WRITE_ACK, because it is waiting for a WRITE-ACK message, for which young delivery is ensured (lemma 1).

The owner stops being owner when it sends a COPY in reply to a valid FORWARDED-WRITE request to a node, and at that time manager M is in state WAIT_WRITE_ACK.

As a result, we have to show that when manager M executes transition 3, node M.new_owner is the owner. In fact, since the only WRITE requests that manager M forwards to the owner are the ones that the requesters have confirmed, when manager M sends a FORWARDED-WRITE request for node n to node M.owner, we have $n = M$.new_owner, and node n is in state WAIT_COPY_WRITE. Moreover, exactly once delivery is ensured for FORWARDED-WRITE messages (lemma 1), and hence the COPY message m that node M.owner sends to node n is valid. Since no other valid COPY message can be sent to node n between the time manager M enters state WAIT_CONFIRM and the time manager M exits state WAIT_WRITE_ACK (node n has confirmed the request, and hence waits for a valid COPY), node n becomes owner when it receives the message m. □

Lemma 6 If a node n sends a request to manager M, then node n eventually receives a reply.

Proof: Since no message is lost, manager M eventually receives the request. Since no message is lost and, according to lemma 5, 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 gets indefinitely blocked, and that if it gets unblocked thanks to a message from the application manager replacing an acknowledgement from a node n, then node n does not get blocked either. This suffices because every request from a node n is sent to manager M, and terminates by an acknowledgement that manager M waits for.

Manager M may get indefinitely blocked when it is in one state in WAIT_INV_ACK, WAIT_CONFIRM, WAIT_WRITE_ACK, and WAIT_READ_ACK, and waits for a message from a node n.

Now we examine the different cases. In state WAIT_INV_ACK, as soon as its failure detector suspects node n, manager M asks the application manager to reply if node n executes no thread. If manager M does not get unblocked because the application manager never replies, then node n executes a thread. Either node n is reachable, in which case it will eventually send the message manager M is waiting for (no message loss), either node n has disconnected, in which case the application manager eventually learns it (completeness of the failure detectors) and decides a restart. In that case, no request from node n is involved.

In state WAIT_CONFIRM, as soon as it suspects node n, manager M asks the application manager to restart the application. Since failure detectors are complete, this eventually happens.

In states WAIT_WRITE_ACK and WAIT_READ_ACK, manager M may block either because node M.owner disconnected before sending the COPY message to node n, either because node n disconnected before sending the acknowledgement. In state WAIT_WRITE_ACK, as soon it suspects node n or node M.owner, manager M asks the application manager to restart the application. Hence, liveness is ensured.

In state WAIT_READ_ACK, as soon as it suspects node n, manager M asks the application manager to reply when node n executes no thread, and as soon as it suspects node M.owner, manager M asks the application manager to restart the application. If node M.owner remains reachable, then for the same reasons as the case of state WAIT_INV_ACK, manager M does not remain blocked. Node n does not get blocked either because it does not wait for a message after having sent the READ-ACK message. If node M.owner disconnects, then manager M eventually learns it and hence the application is restarted.

4 Fault-Tolerant Object Managers and Application Managers

4.1 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[19] and their failure detectors. For object managers, we could use application checkpointing and restart: as soon as a node can not reach an object manager, the application manager restarts the application. However, depending on the mechanism used to generate the object managers IDs,the object managers of all the shared objects of an application may be spread on a large number of nodes, and hence the mean time between failures of object managers may be so short that the application's execution can not progress. For this reason, the object manager's state machine is actively replicated.

As a result, messages sent from managers are duplicated, which already fits in the general assumptions of our protocol.

The determinism of the state machines allow the replicas not to synchronize too much. Indeed, to produce the same outputs, the replicas need only to agree on the order they process their inputs.

Such replication schemes have been studied with several models of failures[19], including byzantine failures, and can be made efficient for little numbers of replicas[3]. With failure detectors in class $\diamond S$, to ensure liveness and that every replica applies the same transitions in the same order despite up to f disconnections, $2f+1$ replicas are needed.

4.2 Mapping Replicas to Nodes

Mapping replicas to nodes involves two problems. 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[18]. The replicas are mapped to the $2f+1$ nodes closest to the key in the ID space, which needs that the neighbour set of nodes includes at least the $2f+1$ closest nodes in the ID space. As a result, to send messages to managers, entities use primitive route_replica.

Second, consequently to disconnections or joins, some nodes may have to be added to or removed from the replica group. In order to preserve coherence of the replicas, the changes to the group are done using a Primary Component Group Membership service[5]. In order to detect when the group membership should be changed, each member of the group monitors its neighbour set. 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 agree to. This should prevent from changing the group membership too frequently because of temporary inconsistencies in the nodes' neighbour sets during reconfigurations. A group membership service similar to the one in the Rampart toolkit[14] can be used to implement the desired service.

5 Conclusion and Future Work

We have presented a coherence protocol for cached copies of volatile objects in structured P2P systems. Our protocol is the first, to our knowledge, that ensures sequential consistency and liveness of shared memory applications on a distributed system despite simultaneous reconfiguration events including failures. Furthermore, to our knowledge, it is the first sequential consistency protocol for cached copies of volatile objects that have been designed for a P2P system.

Our protocol is based on K. Li's protocol with fixed distributed managers, and ensures coherence and liveness despite simultaneous reconfiguration events. Liveness of the application is ensured by transparently locating object managers using the P2P system's DHT, by replicating state machines using the P2P system's neighbourhood information, and by using backward error recovery for thread failures and loss of master copies of shared objects.

Our future work includes optimizing the protocol. In particular, compared to the original protocol of K. Li, requests for copies of shared objects are slown down because the replicas of the object managers must agree on the order they handle requests in. Hence we could benefit from removing the object manager from the critical path, using techniques similar to K. Li's protocol with dynamic distributed managers.

We also plan to design protocols for release consistency in peer-to-peer systems. Release consistency allows multiple writers, and hence can be beneficial if the grain of sharing is smaller than the grain of coherence. Indeed it can be desirable that the grain of coherence be bigger than the grain of sharing if communication links are slow, or to lower the size of metadata for the coherence protocol.

Finally, we plan to implement and evaluate the protocol in a peer-to-peer system similar to Pastry[17].

A Transitions of the Protocol's State Machines

Considering that there is only one object, we describe the messages passed between the object manager and the nodes. To support several objects, the messages sent to the nodes only need to additonally

include the key of the corresponding object manager. We also describe the algorithms applied in the transitions of the object manager's and the nodes' state machines.

The messages of the protocol have the following format:

- READ(requester, seq_num), in which requester is the requester's node ID, and seq_num is the sequence number of the request;
- WRITE(requester, seq_num), in which requester is the requester's node ID, and seq_num is the sequence number of the request;
- WRITE_CONFIRM(requester, seq_num, last_inv), that asks and confirms a WRITE request from node requester of sequence number seq_num, where last_inv is the value of the variable last_inv of the object manager when it has sent the WRITE-CONFIRM message;
- WRITE_INVALIDATE(requester, seq_num, last_inv), in which the fields have the same meaning as for WRITE-CONFIRM messages;
- FORWARDED_WRITE(requester, seq_num, last_inv), in which the fields have the same meaning as for WRITE-CONFIRM messages;
- WRITE_ACK(requester, seq_num), in which requester is the node ID of the node that requested the copy, and seq_num is the sequence number of the last *invalidation* requested by the object manager before forwarding the WRITE-REQUEST to the former owner;
- READ_ACK(requester, seq_num), in which the fields have the same meaning as for READ messages;
- INVALIDATE(seq_num), in which seq_num is the sequence number of the invalidation;
- INVALIDATE_ACK(sender, seq_num), in which sender is the ID of the node that sends the message, and seq_num is the sequence number of the invalidation that the message acknowledges;
- FIRST_TOUCH, for which no field is needed;
- COPY(value, seq_num, last_known_inv), in which value is the value of the object, seq_num is the sequence number of the request for which the COPY message is a reply, and last_known_inv is the last sequence number of invalidation that the owner knows when sending the message.

A.1 Transitions of the Object Manager's State Machine

For a given object, the object manager's state is composed of the following variables:

- state initialized to IDLE;
- pending_req initialized to [] (empty queue);
- owner initialized to _ (not a valid ID);
- new_owner initialized to _ (not a valid ID);
- current_write initialized to 0;
- last_inv initialized to 0;
- waiting_read_ack is a hash table, the keys being node IDs and the values being sequence numbers of READ requests, and is initialized to {} (empty hash table).

The algorithms applied in the transitions are the following:

- Transition 1:

  Condition: WRITE(requester, seq_num)
  pending_req.append(WRITE(requester, seq_num))

- Transition 2:

  Condition: pending_req.not_empty() $\wedge$ owner = _
  new_owner := pending_req.head.requester
  pending_req := pending_req.tail
  send(new_owner, new_owner, COPY_SERVICE, FIRST_TOUCH)

- Transition 3:

  Condition: WRITE_ACK(requester, seq_num) $\wedge$ requester = new_owner $\wedge$ seq_num = last_inv
  owner := new_owner
  copy_set := {}

- Transition 4:

  Condition: READ(requester, seq_num)
  pending_req.append(READ(requester, seq_num))

- Transition 5:

  Condition: pending_req.head = READ $\wedge$ owner $\neq$ _

```
Local req
repeat
  req := pending_req.head
  pending_req := pending_req.tail
  if req.requester ≠ owner then
    copy_set := copy_set ∪ {req.requester}
    waiting_read_ack[req.requester] := req.seq_num
    send(owner, owner, COPY_SERVICE, req)
  end if
until pending_req = [] ∨ pending_req.head ≠ READ
```

- Transition 6:

  Condition: READ_ACK(requester, seq_num)

```
if waiting_read_ack[requester] = seq_num then
  waiting_read_ack.del(requester)
end if
```

- Transition 7:

  Condition: waiting_read_ack = {}
  {Nothing to do}

- Transition 8:

  Condition: $\mathsf{pending_req.head} = \mathrm{WRITE} \wedge \mathsf{owner} \neq _$
  Local req
  req := pending_req.head
  pending_req := pending_req.tail
  new_owner := req.requester
  current_write := req.seq_num
  last_inv + +
  send(new_owner, new_owner, COPY_SERVICE,
  WRITE_CONFIRM(new_owner, req.seq_num, last_inv))

- Transition 9:

  Condition: WRITE_CONFIRM(requester, seq_num, inv_seq)
  $\wedge$requester = new_owner $\wedge$ inv_seq = last_inv
  for all node $\in$ copy_set \ {new_owner} **do**
  send(node, node, COPY_SERVICE, INVALIDATE(last_inv))
  end for

- Transition 10:

  Condition: INVALIDATE_ACK(sender, inv_seq) $\wedge$ inv_seq = last_inv
  copy_set := copy_set \ {sender}

- Transition 11:

  Condition: copy_set $\subseteq$ {new_owner}
  send(owner, owner, COPY_SERVICE,
  FORWARDED_WRITE(new_owner, current_write, last_inv))

- Transition 12:

  Condition: READ(requester, seq_num)
  if requester $\neq$ owner **then**
  if pending_req.not_empty() **then**
  pending_req.append(READ(requester, seq_num))
  else
  copy_set := copy_set $\cup$ {requester}
  waiting_read_ack[requester] := seq_num
  send(owner, owner, COPY_SERVICE, READ(requester, seq_num))
  end if
  end if

- Transition 13:

  Condition: READ(requester, seq_num)
  {Same as transition 4}

- Transition 14:

  Condition: WRITE(requester, seq_num)
  {Same as transition 1}

- Transition 15:

  Condition: WRITE_INVALIDATE(requester, seq_num, inv_seq)
  $\wedge$requester = new_owner $\wedge$ inv_seq = last_inv
  {Nothing to do}

A.2 Transitions of the Node's State Machine

For a given object, the node's state is composed of the following variables:

- key, which is the key of the object manager;
- state initialized to INVALID;
- copy, which contains the copy of the object and is initialized to _ (no value);
- last_known_inv initialized to 0;
- last_req initialized to 0.

The algorithms applied in the transitions are the following:

- Transition 1:
  Condition: WRITE fault
  last_req++
  route(key, OBJECT_MANAGER_SERVICE, WRITE(self, last_req))

- Transition 2:
  Condition: FIRST_TOUCH $\wedge$ (last_req $=1 \vee$ (last_req $=2 \wedge$ last_known_inv $=0$))
  {Authorize read and write accesses to the copy}
  route(key, OBJECT_MANAGER_SERVICE, WRITE_ACK(self, 0))

- Transition 3:
  Condition: READ(requester, seq_num)
  {Authorize only read accesses to the copy}
  send(requester, requester, COPY_SERVICE, COPY(copy, seq_num, last_known_inv))

- Transition 4:
  Condition: FORWARDED_WRITE(requester, seq_num, last_inv) $\wedge$ last_inv $>$ last_known_inv
  last_known_inv := last_inv
  {Forbid any access from threads to the copy}
  send(requester, requester, COPY_SERVICE, COPY(copy, seq_num, last_inv))

- Transition 5:
  Condition: COPY(value, seq_num, last_inv) $\wedge$ seq_num = last_req
  copy := value
  {Authorize read and write accesses to the copy}
  last_known_inv := last_inv
  route(key, OBJECT_MANAGER_SERVICE, WRITE_ACK(self, last_inv))

- Transition 6:
  Condition: READ fault
  last_req++
  route(key, OBJECT_MANAGER_SERVICE, READ(self, last_req))

- Transition 7:
  Condition: FIRST_TOUCH $\wedge$ last_req $=1$
  {Authorize read and write accesses to the copy}
  route(key, OBJECT_MANAGER_SERVICE, WRITE_ACK(self, 0))

- Transition 8:

  Condition: $\text{COPY}(\mathsf{value}, \mathsf{seq_num}, \mathsf{last_inv}) \wedge \mathsf{seq_num} = \mathsf{last_req}$
  $\mathsf{copy} := \mathsf{value}$
  {Authorize only read accesses to the copy}
  $\mathsf{last_known_inv} := \mathsf{last_inv}$
  $\mathsf{route}(\mathsf{key}, \mathsf{OBJECT_MANAGER_SERVICE}, \text{READ_ACK}(\mathsf{self}, \mathsf{last_req}))$

- Transition 9:

  Condition: $\text{INVALIDATE}(\mathsf{seq_num}) \wedge \mathsf{seq_num} > \mathsf{last_known_inv}$
  {Forbid any access to the copy}
  $\mathsf{last_known_inv} := \mathsf{seq_num}$
  $\mathsf{route}(\mathsf{key}, \mathsf{OBJECT_MANAGER_SERVICE}, \text{INVALIDATE_ACK}(\mathsf{self}, \mathsf{seq_num}))$

- Transition 10:

  Condition: WRITE fault
  {Same as transition 1}

- Transition 11:

  Condition: WRITE fault
  {Same as transition 1}

- Transition 12:

  Condition: $\text{FORWARDED_WRITE}(\mathsf{requester}, \mathsf{seq_num}, \mathsf{last_inv})$
  $\wedge \mathsf{requester} = \mathsf{self} \wedge \mathsf{last_inv} > \mathsf{last_known_inv}$
  {Authorize read and write accesses to the copy}
  $\mathsf{last_known_inv} := \mathsf{last_inv}$
  $\mathsf{route}(\mathsf{key}, \mathsf{OBJECT_MANAGER_SERVICE}, \text{WRITE_ACK}(\mathsf{self}, \mathsf{last_inv}))$

- Transition 13:

  Condition: $\text{FORWARDED_WRITE}(\mathsf{requester}, \mathsf{seq_num}, \mathsf{last_inv}) \wedge \mathsf{last_inv} > \mathsf{last_known_inv}$
  {Same as transition 4}

- Transition 14:

  Condition: $\text{FORWARDED_WRITE}(\mathsf{requester}, \mathsf{seq_num}, \mathsf{last_inv})$
  $\wedge \mathsf{requester} \neq \mathsf{self} \wedge \mathsf{last_inv} > \mathsf{last_known_inv}$
  $\mathsf{last_known_inv} := \mathsf{last_inv}$
  {Forbid any access to the copy}
  $\mathsf{send}(\mathsf{requester}, \mathsf{requester}, \mathsf{COPY_SERVICE}, \text{COPY}(\mathsf{copy}, \mathsf{seq_num}, \mathsf{last_inv}))$

- Transition 15:

  Condition: $\text{INVALIDATE}(\mathsf{seq_num}) \wedge \mathsf{seq_num} > \mathsf{last_known_inv}$
  {Same as transition 9}

- Transition 16:

  Condition: $\text{INVALIDATE}(\mathsf{seq_num}) \wedge \mathsf{seq_num} > \mathsf{last_known_inv}$
  {Same as transition 9}

- Transition 17:

  Condition: $\text{READ}(\mathsf{requester}, \mathsf{seq_num})$
  $\mathsf{send}(\mathsf{requester}, \mathsf{requester}, \mathsf{COPY_SERVICE}, \text{COPY}(\mathsf{copy}, \mathsf{seq_num}, \mathsf{last_known_inv}))$

- Transition 18:
  Condition: READ(requester, seq_num)
  {Same as transition 17}

- Transition 19:
  Condition: WRITE fault
  {Same as transition 1}

- Transition 20:
  Condition: WRITE fault
  {Nothing to do}

- Transition 21:
  Condition: WRITE fault
  {Nothing to do}

- Transition 22:
  Condition: READ fault
  {Nothing to do}

- Transition 23:
  Condition: READ fault
  {Nothing to do}

- Transition 24:
  Condition: COPY(value, seq_num, last_inv)
  $\wedge\neg$((state = WAIT_PAGE_READ $\vee$ state = WAIT_PAGE_WRITE)
  $\wedge$seq_num = last_req)
  route(key, OBJECT_MANAGER_SERVICE, READ_ACK(self, seq_num))

- Transition 25:
  Condition: INVALIDATE(seq_num) $\wedge$ seq_num > last_known_inv
  route(key, OBJECT_MANAGER_SERVICE, INVALIDATE_ACK(self, seq_num))

- Transition 26:
  Condition: WRITE_CONFIRM(requester, seq_num, last_inv)
  if (state = WAIT_WRITE $\vee$ state = WAIT_COPY_WRITE) $\wedge$ seq_num = last_req **then**
  route(key, OBJECT_MANAGER_SERVICE, WRITE_CONFIRM(self, seq_num, last_inv))
  else
  route(key, OBJECT_MANAGER_SERVICE, WRITE_INVALIDATE(self, seq_num, last_inv))
  end if

References

[1] 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.

[2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*. USENIX, December 2002.

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

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

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

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP'01)*, pages 202–215. ACM Press, 2001.

[7] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II (IPTPS 2003)*, volume 2735 of *Lecture Notes in Computer Science*, pages 33–44. Springer, February 2003.

[8] 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.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[11] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II (IPTPS 2003)*, volume 2735 of *Lecture Notes in Computer Science*, pages 21–32. Springer, February 2003.

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

[13] 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.

[14] Michael K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938, pages 99–110. Springer-Verlag, Berlin Germany, 1995.

[15] 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.

[16] Louis Rilling and Christine Morin. A coherence protocol for cached copies of volatile objects in peer-to-peer systems. Publication interne 1581, IRISA, December 2003.

[17] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[18] 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. ACM Press, 2001.

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

[20] 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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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