HAL
open science

SAM: Self* Atomic Memory for P2P Systems

Emmanuelle Anceaume, Maria Gradinariu, Vincent Gramoli, Antonino Virgillito

To cite this version:

Emmanuelle Anceaume, Maria Gradinariu, Vincent Gramoli, Antonino Virgillito. SAM: Self* Atomic Memory for P2P Systems. [Research Report] PI 1717, 2005, pp.23. inria-00000115

HAL Id: inria-00000115

https://inria.hal.science/inria-00000115v1

Submitted on 17 Jun 2005

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.

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1717

SAM: SELF* ATOMIC MEMORY FOR P2P SYSTEMS

EMMANUELLE ANCEAUME MARIA GRADINARIU
VINCENT GRAMOLI ANTONINO VIRGILLITO

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

SAM: Self* Atomic Memory for P2P Systems

Emmanuelle Anceaume Maria Gradinariu
Vincent Gramoli Antonino Virgillito

Systèmes communicants
Projet Adept

Publication interne n1717 — June 2005 — 23 pages

Abstract: We propose an implementation of *self-adjusting and self-healing atomic memory* in highly dynamic systems exploiting peer-to-peer (p2p) techniques. Our approach, named *SAM*, brings together new and old research areas such as p2p overlays, dynamic quorums and replica control. In SAM, nodes form a connected overlay. To emulate the behavior of an atomic memory we use intersected sets of nodes, namely *quorums*, where each node hosts a replica of an object. In our approach, a quorum set is obtained by performing a deterministic traversal of the overlay. The SAM overlay features self-* capabilities: that is, the overlay self-heals on the fly when nodes hosting replicas leave the system and the number of active replicas in the overlay dynamically self-adjusts function of the object load. In particular, SAM pushes requests from loaded replicas to less solicited replicas. If such replicas do not exist, the replicas overlay self-adjusts to absorb the extra load without breaking the atomicity. We propose a distributed implementation of SAM where nodes exploit only a restricted local view of the system, for the sake of scalability. We provide a complete specification of our system and prove that it implements object atomicity.

Key-words: Distributed Systems, Atomic Memory, Self-* Systems, Quorum, Scalability, Fault Tolerance

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Mémoire atomique auto-reconfigurable pour systèmes P2P

Résumé : Ce rapport présente une mémoire atomique auto-ajustable et auto-réparante en systèmes hautement dynamiques. Nous proposons une implémentation de celle-ci basée sur l'utilisation de techniques égales-à-égales (p2p). Cette solution, appelée *SAM*, rassemble des thématiques de recherches aussi bien anciennes que récentes telles que les couches de communication p2p, les quorums dynamiques et la réplication contrôlée. Les nœuds de SAM forment un sur-graphe connecté. Une copie de chaque objet est répliquée à différents nœuds, appelés *réplicas*. Afin d'assurer l'atomicité de ces objets, nous utilisons des *quorums*, ensembles intersectés de réplicas. Ces quorums sont obtenus via une traversée déterministe effectuée au sein du graphe de communication. De plus ce graphe possède des propriétés auto-* : Celui-ci s'auto-répare à la volée lorsque des réplicas quittent le système et le nombre de réplicas s'auto-ajuste en fonction de la charge. Plus particulièrement, SAM réparti automatiquement la charge en distribuant les requêtes effectuées sur des nœuds surchargés à d'autres nœuds. Si tous les nœuds sont surchargés alors le sur-graphe s'auto-ajuste pour absorber la charge induite en garantissant l'atomicité. Dans ce rapport nous proposons une implementation distribuée de SAM où chaque nœud ne possède qu'une connaissance locale du système, permettant ainsi son utilisation à grande échelle. Nous spécifions formellement cet algorithme et prouvons qu'il satisfait la propriété d'atomicité des objets.

Mots clés : Systèmes répartis, Mémoire atomique, Systèmes Auto-*, Quorum, Passage à l'échelle, Tolérance aux défaillances

1 Introduction

The real notoriety of peer-to-peer file sharing guarantees the subsequent success of p2p systems in commercial applications. However, in order to move further, the p2p community needs to focus on designing fundamental abstractions that offer strong computational guarantees. Atomic memory is a basic service in distributed computing that offers a persistent storage with linearizable read/write semantics. This service has a broad Internet-scale applications. Consider e-auctions, for example. The atomic memory could be used by each auctioneer to write its bid and read the others bids. Alternatively, the distributed on-line booking needs an atomic memory in order to record in a persistent manner the state of the booking process.

Designing atomic memory in p2p systems faces several problems. p2p systems are by their nature ad-hoc distributed systems without any organization or centralized control. Unlike classical distributed systems, p2p systems encompass processes (peers) that experience highly dynamic behaviors including spontaneous join and leave or change in their local connections. The high dynamism of the network has a tremendous impact on data availability. The use of classical distributed computing solutions, like replication, for example, introduces an extra cost related to: (1) maintaining a sufficient number of replicas despite frequent disconnections; and (2) maintaining the consistency among the replicas. The former problem can be solved using *self-healing* techniques while the latter one finds solutions in the use of *dynamic quorums* (intersecting sets).

Another issue posed by the dynamism of the network is the replica stress (load). An inadequate number of replicas may have important impact on the replica access latency since the access latency increases with the access rate. Moreover, due to limitations of the local buffers size a non negligible fraction of replica accesses might be lost. Consequently, the number of replicas should spontaneously adjust to the access rate.

1.1 Related Works

Starting with Gifford's weighted votes [9], quorum systems [3, 14, 6, 27] have been widely used to provide consistency. Several quorum-based approaches have been used to provide mutual exclusion [20] or shared memory emulation [5]. Recently, quorum-based implementations of atomic memory for dynamic systems have been proposed in [16, 7, 10]. All these works have a common design seed — they use reconfigurable quorum systems, work pioneered by Herlihy [12] for static distributed systems. In cite [19, 8] the authors showed that using two quorums systems concurrently preserves atomicity. This result has been latter exploited in the implementation of the reconfigurable quorum systems for highly dynamic systems. That is, periodically the system proceeds to modifications of the current quorums set (referred as configuration). This reconfiguration process is handled in [16, 10] by using Paxos [15]. The Paxos consensus algorithm serves to agree on a total order of the configurations. Alternatively, a restricted reconfiguration process is used in [7] in order to cope with dynamic operational statistics in ad-hoc networks. That is, during periods of time in which the number of read operations exceeds the number of write operations the authors advocate for the use of fast read quorums and slow write quorums. When the statistics of the operations change then the system reverses its strategy via reconfiguration. In [7] the reconfiguration process does not need consensus since the system is limited to the use of a small finite set of configurations.

In this work we follow an alternative approach for implementing atomic memory in dynamic systems stared by the recent achievements in the context of dynamic quorums and p2p overlays. Dynamic quorums have been mainly investigated in [24, 2, 21]. Naor and Wieder, [24] sought solutions for deterministic quorums using dynamic paths in a planar overlay [22]. Simultaneously, probabilistic quorums were proposed by Abraham and Malkhi [2] based on an overlay designed as a dynamic approximation of De Bruijn graphs [1]. Recently, in [27] the authors discuss the impact of dynamism on the multi-dimensional quorum systems for read-few/write-many replica control protocols. They briefly describe strategies for the design of multi-dimensional quorum systems that combine local information in order to deal with frequent join and leaves of replicas and quorum sets caching in order to reduce the access latency. ANDOr strategies [23] are studied in [21] in order to implement fault-tolerant storage in dynamic environment.

1.2 Contributions

In this report we propose a modular construction of an atomic memory for dynamic systems with self-adjusting and self-healing capabilities. Our approach brings together several new and old research areas exploiting the best of these worlds: p2p overlays, dynamic quorums and replica control.

The architecture of our system is composed of three interconnected modules each being designed to serve for a specific task (i.e. data availability, linearizability or load balancing of replicated data).

To ensure data availability despite the system dynamism, we replicate data among several nodes. Replicas of the same object define a torus overlay (similar to CAN [25]) proved efficient in the design of quorum systems ([13, 23]). A specific module in our solution, *Adjuster*, serves to heal the overlay when replicas fail. Moreover, this module dynamically adjusts the size of the replicas overlay function of the replicas stress.

To emulate the behavior of an atomic memory we use intersected sets of nodes, namely *quorums*. In our approach, a quorum set is sampled from a deterministic overlay traversal, encapsulated in the *Traversal* module. To ensure atomicity—i.e. linearizability—of read/write operations we perform appropriate read and write traversal strategies. The particularity of our approach is the use of a single round-trip communication phase for read operations. This improves the efficiency of the atomic memory when read operations are frequent compared with write operations. Moreover, when a pick of requests occurs we overlap operations without breaking atomicity. We use Input/Output Automata [17] to formally specify our algorithm and prove its atomicity property.

Finally, in order to balance the load of the system we propose a strategy, encapsulated in the *Thwarter module*, that aims at pushing requests from loaded nodes to less solicited nodes. If such nodes do not exist new replicas are added to the system without breaking the atomicity by the mean of the Adjuster module.

We have exploited p2p techniques in order to provide an on demand atomic memory. In response to the request access rate, the atomic memory is expanded or reduced to fit the demand while preserving reasonable probe-complexity. The p2p techniques we use allow to augment the atomic memory with self* features using only constant size local information (lightweight reconfiguration). That is, each replica only maintains information related to its neighborhood in the replicas overlay and clients need to know only one node in this overlay. Moreover, unlike solutions based on reconfigurations clients or replicas owners do not have to be aware of the reconfiguration process. In conclusion, our work can be seen as a hybrid between the reconfiguration based systems and strategic adaptive systems. We use lightweight reconfiguration in order to achieve the self-healing and self-adjusting properties and adaptive strategies in order to sample read and write quorums and to balance the replica stress.

The report is organized as follows. The system model is proposed in Section 2. An overview of SAM is presented in Section 3. SAM specification appears in Section 4 and the proof of SAM's atomicity is given in Section 5. Finally, in Section 6 we conclude and present some future research topics.

2 Model

In this section we present the model used in SAM. First we emphasize the dynamic aspect of our model, the communication pattern used and we present the overlying communication graph. Second, we restate the atomicity definition proposed by Lynch.

2.1 Dynamic System Model

All modern applications in dynamic distributed systems are based on *the principle of data independence* — the separation of data from the programs that use the data. This concept was first developed in the context of database management systems. In the following we consider a dynamic system $\mathcal{DS}$ as the tuple $\mathcal{DS} = (I, X)$, where I is a set of finite, yet unbounded node identifiers, and X is an unbounded universe of shared data, referred in the following as objects.

The physical network is described by a weakly connected graph. Its nodes represent processes of the system and its links represent established communication links between processes. The graph is referred in the following as the communication graph. The communication graph is subject to frequent and unpredictable changes: nodes can leave or join the system arbitrarily often, and they can fail temporarily (transient faults) or permanently (crash failures).

Each object has an unique owner (the node hosting the object) and may be replicated at the other nodes. The only actions executed on each object are reads, writes and replicate. Read and write operations are defined by two type of traversals. Thus each of them consists in probing a set of nodes by traversing the logical overlay described below. By abuse of notation, we refer to a read or a write operation as respectively, a read or a write traversal.

We consider the network plus the data stored in the network represented by a logical multi-layer overlay, each logical layer l being a weakly connected graph, also referred to as the logical communication graph at layer l. In order to connect to a particular layer l, a node executes an underlying connection protocol. A node $i \in I$ is called *active* at a

layer l if there exists at least one node j which is connected at l and aware of i. The set of logical *neighbors* of a node i at a layer l is the set of nodes j such that the logical link (i, j) is up (i and j are aware of each other). Notice that a node i may belong to several layers simultaneously. Thus, i may have different sets of neighbors at different logical layers. Can, Pastry or Chord ([25, 26, 28]) are typical logical overlays using DHTs as design principle.

Replicas of an object share a same logical overlay, organized in a torus topology (as for example CAN [25]). Basically, a 2-dimensional coordinate space $[0, 1) \times [0, 1)$ is shared by all the replicas of an object. Thus we say that the bounds b of the system are given by the minimal abscissa $b.xmin = 0$, and ordinate $b.ymin = 0$ and the maximal abscissa $b.xmax = 1$ and ordinate $b.ymax = 1$. Each replica $i \in I$ has an exclusive responsibility region in this space. Likewise its region is given by its $[i.xmin, i.xmax) \times [i.ymin, i.ymax)$ interval product.

2.2 Self* Atomic Memory

In this work we emulate an atomic memory with self* capabilities. Atomicity is often defined in terms of an equivalence with a serial memory. In the following we adopt the definition proposed in [16].

Definition 1 (Atomicity) *Let $\mathcal{DS} = (P, X)$ be a dynamic system. If all the read and write operations that are invoked complete, then the read and write operations for object x can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied:*

1. *The partial order is consistent with the external order of the invocations and responses, that is, there does not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$;*
2. *All write operations are totally ordered and every read operation is ordered with respect to all the writes;*
3. *Every read operation ordered after any write returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns the initial value of the object.*
4. *No operation has infinitely many other operations ordered before it;*

In order to be operational in a dynamic environment, two additional properties are required from an atomic memory: *self-healing* and *self-adjusting*. Self-healing aims to ensure the availability of an object whenever failures occur while self-adjusting aims to expand or restraint the number of replicas function of the access rate.

3 SAM Overview

SAM aims at emulating an atomic memory on top of a replicated system with self-adjusting and self-healing capabilities. In this section we gives an informal description of SAM. We present how SAM copes with dynamism, how operations resort to quorums for guaranteeing atomicity, and how SAM auto-adjust in case a non-willing load is detected.

3.1 Dealing with Dynamism

The entrance and departure of a replica dynamically changes the decomposition of the regions. These regions are rectangles in the plane. Replicas owners of adjacent regions are called neighbors in the overlay and are linked by virtual links. The overlay has a torus topology in the sense that the zones over the left and right (resp. upper and lower) borders are neighbors of each other. Initially, only the owner of the object is responsible for the whole space. The bootstrapping process pushes a finite, bounded set of replicas in the network. These replicas are added to the overlay using well-known strategies [25, 24]: the owner of the object specifies randomly chosen points in the logical overlay, and the zone in which each new replica falls is split in two. Half the zone is left to the owner of the zone, and the other half is assigned to the new replica. In the following we omit a more detailed description of the bootstrapping process. Note that an interesting point to explore here is the introduction of efficient incentive mechanisms to motivate nodes to host replicas (i.e. to be part of the atomic memory). Techniques from game theory or mechanism theory can be used to this end, however this topic is beyond the scope of this paper.

3.2 Quorum Based Operations

Replicas are accessed by clients through read and write operations on the object. Each read operation consists in traversing the overlay following a horizontal trajectory that wraps all the overlay. All the zones (more precisely, all the replicas identified by these zones) that intersect this traversal define a *consultation quorum*. Each write operation consists in traversing the overlay following a horizontal trajectory and then a vertical one. All the zones (more precisely, all the replicas identified by these zones) that intersect this traversal define a *propagation quorum*. Please refer to Definitions 2 and 3 for a formal description.

Definition 2 (Consultation Quorum Q_c) *A consultation quorum $Q_c \subset I$ is a set of nodes, such that*

- $\bigcup_{\forall j \in Q_c} \{[j.xmin, j.xmax)\} = [b.xmin, b.xmax)$
- $\bigcap_{\forall j \in Q_c} \{[j.xmin, j.xmax)\} = \emptyset$
- $\exists i \in Q_c, \forall j \in Q_c, i.ymin + (i.ymax - i.ymin/2) \in [j.ymin, j.ymax)$

Definition 3 (Propagation Quorum Q_p) *A propagation quorum $Q_p \subset I$ is a set of nodes, such that*

- $\bigcup_{\forall j \in Q_p} \{[j.ymin, j.ymax)\} = [b.ymin, b.ymax)$
- $\bigcap_{\forall j \in Q_p} \{[j.ymin, j.ymax)\} = \emptyset$
- $\exists i \in Q_p, \forall j \in Q_p, i.xmin + (i.xmax - i.xmin/2) \in [j.xmin, j.xmax)$

Each read quorum intersects each write quorum. When object x is written, it is written at each replica of a writing quorum. When the value of object x is searched, all the replicas of a read quorum are queried.

Theorem 3.1 *For any consultation quorum Q_c, and propagation quorum Q_p, the following intersecting property holds: $Q_c \cap Q_p \neq \emptyset$.*

Proof. The result follows trivially from definitions 2 and 3. For all pair Q_c, Q_p, the replica responsible for point $(i.xmin + (i.xmax - i.xmin/2), i.ymin + (i.ymax - i.ymin/2))$ belongs to $Q_c \cap Q_p$. □

3.3 Load Balancing

SAM starts with a read/write request from a client. A client submits a request to one of the replicas of the overlay. This replica is referred as the initiating replica. Upon receipt of a read (resp. write) request, the initiating replica does not immediately initiate a read/write traversal but it rather enqueues the request. All replicas periodically scan their queues to pick the requests for which a traversal is initiated. The strategy to pick these requests is as follows: A write traversal is initiated only for the most recently enqueued write request (if any), while a read traversal is initiated for one of the enqueued read request (if any). Old write operations can be safely discarded (no write traversals are initiated for them) as they will not influence anymore the state of the object. There is no such constraint for a read. Once the traversals are initiated, the initiating replica empties its queue, after having kept track of all the read/write requests to later return the status of the read operation and the write operation to the requesting clients. Despite this request aggregation strategy allowing to reduce the requests that are actually served, the number of enqueued requests at an initiating replica can still grow very fast, incurring in a local overload. To prevent this, if the length of the queue is above a predefined threshold, then received read/write requests are forwarded to another replica that is free enough. The search of a non-overloaded replica consists in visiting the overlay along a diagonal line. If such a replica is found, then it becomes the initiating replica for these requests, otherwise the size of the replica overlay (that is, the quorum system) is expanded for supporting new requests. Alternatively, when the queue of a replica, it leaves voluntarily the overlay. Thus, the size of the replica overlay is shrinks in order to adjust the current system load.

When a replica leaves (voluntary or not), the zone is locally healed by relying on a strategy similar to the one proposed in CAN. On the other hand, joins are triggered by SAM as follows: a replica is inserted within the quorum system only when it is required (i.e., for expansion purpose).

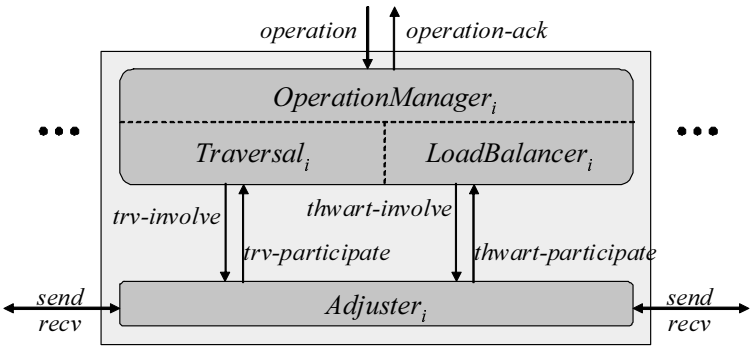

Figure 1: Overview of SAM at node i for a single object

3.4 The Modular Approach

SAM is specified in Input/Output Automata (IOA) Language [17, 18] and is structured as the composition of four main automata:

- $OperationManager_{x,i}$ automaton
- $Adjuster_{x,i}$ automaton
- $CommunicationLink_{x,i,j}$ automaton
- $Joiner_{x,i}$ automaton

Where $i \in I$ and $x \in X$ is the considered object.

The $OperationManager_{x,i}$ has two goals. That is its transitions are divided in two different sets, each serving a specific objective, *traversing* of *thwarting*: we refer to the transitions sets as respectively the $Traversal_{x,i}$ and the $LoadBalancer_{x,i}$. For the sake of simplicity in the IOA code, we merged those two sets of transitions in one $OperationManager_{x,i}$ automaton, since the states used by both are quite identical. Briefly, the $LoadBalancer_{x,i}$ transitions scans the overlay to identify non overloaded replicas. The $Traversal_{x,i}$ automaton is in charge of maintaining the consistency of the overlay applying appropriate strategies for executing linearizable operations on the replicas. The $Adjuster_{x,i}$ handles the expansion or shrink of the quorum system whenever requested by the $LoadBalancer_{x,i}$ automaton, and the departure of non responding replicas. The main function of this automaton is to assign logical responsibility zones to physical replicas and maintain this correspondence consistent.

The $CommunicationLink_{x,i,j}$ receives messages after a $\mathsf{send}_{i,j}$ output event of the $Adjuster_{x,i}$ and send them to the appropriate target j by the mean of a $\mathsf{recv}_{i,j}$ output event. Finally, the $Joiner_{x,i}$ works roughly as follow: it is contacted by a joining node. After the node receives an acknowledgment from this automaton, it is considered as joined. Notice at this time the node is part of the system but might not be a replica yet. We do not specify $CommunicationLink_{x,i,j}$ and $Joiner_{x,i}$ automata here, but we rather focus on the $Traversal_{x,i}$, the $LoadBalancer_{x,i}$ and the $Adjuster_{x,i}$, all specified in Section 4.

In the remaining of the report, we restrict our attention to only one object x. Thus, the x subscript is omitted. Relationship among the three automata in terms of input/output actions is depicted in Figure 1.

4 SAM in details

We present here the formal specification of the SAM algorithm. For this purpose we use the IOA language which is roughly based on precondition-effect actions that specify each component automaton behavior. First we specify the $Traversal$ and the $LoadBalancer$ modules as part of a single $OperationManager$ automaton that handles operations, then we present a lower-level module, called the $Adjuster$ automaton, handling the overlay.

4.1 The Traversal

The traversal is in charge of maintaining the consistency of the replicas applying appropriate strategies for executing linearizable operations via read or write quorums.

In [5, 16] a read operation is realized in two phases. The first one aims at gathering the tag and the value of the last write, and the second to propagate the latest (possibly new) tag-value pair to a write quorum. GeoQuorums [7] aims at reducing the cost of read operations by allowing their execution in only one phase (i.e. the consultation phase). This is achieved by including an additional phase in the write operation, namely the *confirmation* phase, in which the initiator of a write sends a specific confirmation message when the write operation is completed.

Similar to GeoQuorums, SAM aims at improving the efficiency of the atomic memory by maintaining a single phase for read operations. However, one-phase read operation might violate the linearizability. That is, a read operation executed concurrently with a write operation may consult a fresh value not completely propagated, while a latter read may consult an old value (see Figure 2).

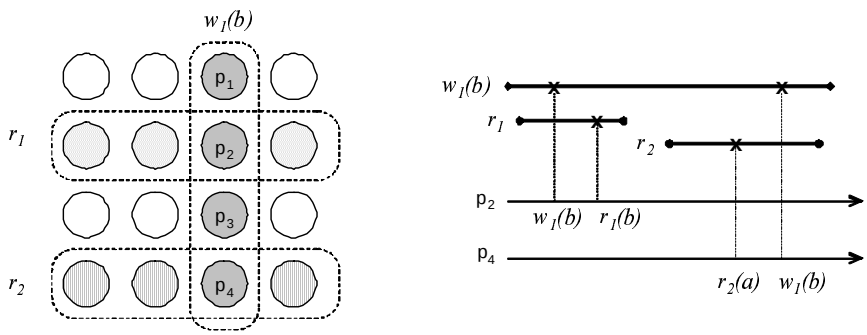

Figure 2: Atomicity Violation

This violates the atomicity definition (see Definition 1). Since the problem comes from the early termination of a read operation that consults a fresh value which is still in a propagation phase, we propose to let this propagation terminate before allowing the read to terminate. That is, either another read operation consults this fresh written value or it is ordered before the first one.

The write strategy in SAM is composed of a *consultation* and a *propagation* phase, while the read strategy includes only the consultation phase. In the consultation phase, the overlay is traversed from one side to another (for example west to east) and all the nodes encountered are requested for the object value and timestamp. Finally, the most up-to-date value is returned. In the propagation phase, the overlay is traversed in two orthogonal directions with respect to consultation. This guarantees that any write and read quorums intersect.

Propagation proceeds in both directions so that each replica is visited twice in this phase: the first time the object is locked, preventing concurrent reads to get a stale value, while the second time the lock is released, indicating the completion of the write operation. In dynamic systems an object may be locked forever due to unforeseen leaves. We deal with this problem by assuming the use of a leasing strategy [11]. We assume without loss of generality that in the consultation phase the overlay is traversed from west to east, while in the propagation phase it is traversed toward both north and south directions. Figure 3 shows the operation phases. In the following we detail the states and the transitions of the Traversal automaton.

4.1.1 States

The state variables of the $OperationManager$ are decribed for node i in Figure 4. First of all, each replica maintains a tag and a value of the object by the mean of the tag and val fields. These fields are updated during operation at some replica depending on their quorum belongingness. The trv record fully describes a traversal. It contains some identifying subfields such as its identifier tid, its $type$ which indicates if the operation associated is a read or a write, the node that requested this operation namely the requester $rqstr$, and the replica that decided to start the traversal called the initiator $intr$. Other trv's subfields are dynamically changed when the traversal is pending: the tag and val are updated during the consultation phase, the $dirs$ set is the direction to which the traversal has to be sent ($\{E\}$ at the beginning and then possibly $\{N, S\}$ if a propagation starts) and the $phase$ indicates the traversal progression.

Finally the $batch$ subfield is essentially used by the $LoadBalancer$ as indicated in Subsection 4.2 for overlapping some traversals.

The traversal aims at contacting a set of replicas among the quorum system (it literally traverses the quorum system). That is messages are sent from neighbor to neighbor, starting at the initiator replica $intr$. Similar messages are also used by the thwart mechanism that is described in the Subsection 4.2. The messages sent during the traversal contain the traversal identifier tid, type $type$, intitiator $intr$, but also information about where it has to be sent (regarding to dir and $line$). The starter str is the node that starts the thwart mechanism (see Subsection 4.2 for details on this mechanism). The str field equals $-$ in case of a traversal message.

There are other fields that are related to the Traversal. The boolean $failed$ field indicates whether the current node is crashed or not, the boolean $replica$ indicates if the node is a replica of the object. Moreover, the $pending\text{-}prop$ field is used to specify which traversal propagation is pending when the current read traversal is done. That is, it is a mapping from the read traversal to all the propagation traversals it encounters. This field and the $locked$ field are used to lock the read traversal until some propagation phases terminate for guaranteeing linearizability.

Finally the accumulator field, namely acc, indicates each phase termination. If the initiator of traversal τ has received one of the traversal messages it has sent, it knows that all a quorum has been contacted. In that case, $acc[\tau] \neq \emptyset$. For instance, when the initiator receives back a traversal τ message coming from the E direction, it knows that a consultation quorum of replica has been contacted, hence $\{E\} \subseteq acc[\tau]$ and the consultation phase is finished. Likewise, if it receives back two messages of a traversal it has initiated and coming from the N and S directions, then $\{N, S\} \subseteq acc[\tau]$ which means that the propagation phase is complete.

4.1.2 Transitions

The transitions of the Traversal automaton are described for node i in Figure 5. A read or write traversal starts at node i in the consultation phase as a result of a $\mathsf{operation}_i$ input event. The traversal starts with the $\mathsf{operation}_i$ event choosing a new unique identifier for it and initializing all the variables. The traversal proceeds with messages being sent by a node i to one of its east neighbors j, through a $\mathsf{send}_{i,j}$ event. Such an event is triggered only if i is not locked/freeze or the current operation is a write. Otherwise the action is blocked until one of these conditions occur.

When a node j receives a traversal message from the $Adjuster_j$ automaton by a $\mathsf{trv\text{-}participate}_j$ input event, it checks whether it is the initiator of this traversal. If he is not, j simply copies traversal information to forward the message in the same direction, and updates its $\langle tag, value \rangle$ pair with the one received (when the received value is more recent than the one stored locally). Since the topology is a torus, contacting successive neighbors in the same sense involves obviously to re-contact the initiator at some point. The acc field is used by the initiator to stop a

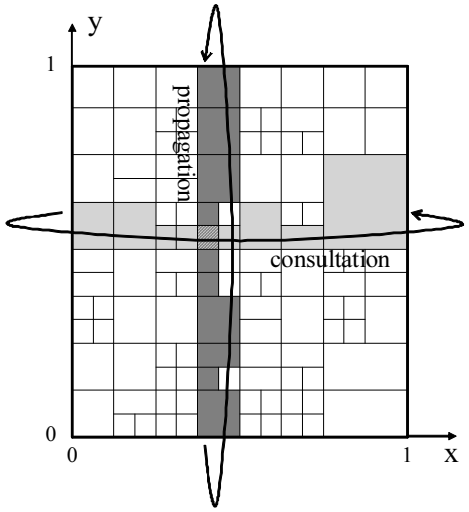

Figure 3: The Traversal

Domains:

I, the set of node identifiers.

$TId \in \mathbb{N} \times I$, the set of traversal identifiers.

V, the set of all possible values for an object.

States:

trv a record with fields
- $tid \in TId$
- $type \in \{\mathsf{read}, \mathsf{write}\}$
- $rqstr \in I$
- $intr \in I$
- $val \in V$
- $tag \in \mathbb{N} \times I \times \mathbb{N}$
- $phase \in \{\mathsf{idle}, \mathsf{starting}, \mathsf{waiting}, \mathsf{ending}\}$
- $dirs \subseteq \{N, S, E\}$
- $batch$, an ordered set of traversals trv

m a record with fields
- $tid \in TId$
- $type \in \{\mathsf{read}, \mathsf{write}\}$
- $intr \in I$
- $str \in I$
- $dir \in \{N, S, E\}$
- $line \in \mathbb{N} \times \mathbb{N}$

tag a record with fields
- $ct \in \mathbb{N}$
- $id \in I$
- $index \in I$
- $val \in V$

$failed$ a boolean

$replica$ a boolean

$queue$ an ordered set of traversals trv

$treating$ a set of traversals trv

$pending\text{-}props$ a set of traversals trv

$clock \in \mathbb{R}^{>0}$

$treat\text{-}time \in \mathbb{R}^{>0}$

$treat\text{-}fqcy \in \mathbb{R}^{>0}$

$shrink\text{-}time \in \mathbb{R}^{>0}$

$shrink\text{-}fqcy \in \mathbb{R}^{>0}$

$threshold \in \mathbb{R}^{>0}$

$fwd \in TId$

$starter \in I$

$order\text{-}expand$ a boolean

$r\text{-}access \in \mathbb{N}$

$w\text{-}access \in \mathbb{N}$

acc a mapping from TId to $\{N, S, E\}$

$locked$ a mapping from TId to TId

Signature:

Input:
- $\mathsf{operation}(type, v)_{j,i},\ i \in I,$ $type \in \{\mathsf{read}, \mathsf{write}\}, v \in V$
- $\mathsf{trv\text{-}participate}(m)_i,\ i \in I$
- $\mathsf{thwart\text{-}participate}(m)_i,\ i \in I$
- $\mathsf{update\text{-}adj\text{-}om}(t, v, pp)_i,\ i \in I,$ $t \in T, v \in V, pp \in TId \times \{N, S, E\}$

Internal:
- $\mathsf{thwart}_i,\ i \in I$
- $\mathsf{trv\text{-}cons}_i,\ i \in I$
- $\mathsf{trv\text{-}prop}_i,\ i \in I$
- $\mathsf{set\text{-}lock}_i,\ i \in I$

Output:
- $\mathsf{trv\text{-}involve}(m)_i,\ i \in I$
- $\mathsf{thwart\text{-}involve}(m)_i,\ i \in I$
- $\mathsf{shrink}_i,\ i \in I$
- $\mathsf{expand}(ra, wa)_i,\ i \in I, ra, wa \in \mathbb{N}$
- $\mathsf{operation\text{-}ack}(v)_{i,j},\ i \in I, v \in V$
- $\mathsf{update\text{-}om\text{-}adj}(t, v, pp)_i,\ i \in I, t \in T, v \in V,$ $pp \in TId \times \{N, S, E\}$

Figure 4: $OperationManager_i$ automaton: Signature and states

traversal: when the initiator writes the traversal direction in acc this completes the consultation phase, triggering the operation-ack action for a read operation or the trv-prop action for a write.

In the propagation phase, the traversal is made in both N and S directions, by the trv-involve and trv-participate events. The write operation completes by sending a operation-ack output event. The only difference with the previous phase is that a node sets a local lock to prevent a concurrent read of a non updated value. The lock is set at a process by the first message receipt and unlocked by the second message receipt. Note the presence of the failed input action. This action is trigged by the environment and aims at indicating that a crash has occured. That is a $failed$ flag, initially false is set to true and any other action is disabled.

4.2 The Load Balancer

The $LoadBalancer$ receives the read/write requests from clients. If the local load induced by those requests is not too high, then it triggers a traversal (i.e., activates the traversal automaton). Otherwise the request cannot be treated because of an overload, the load balancer automaton invokes a *thwart* process to find a suitable replica. The *thwart* process checks the overlay along a diagonal trajectory until finding a non overloaded replica. If the quorum system

Internal $\mathsf{trv\text{-}cons}_i$
Precondition:
$\neg failed \wedge replica$
$queue \neq \emptyset$
$clock \geq treat\text{-}time$
$W = \{w \in queue : w.type = \mathsf{write}\}$
$R = \{r \in queue : r.type = \mathsf{read}\}$
Effect:
if $W \neq \emptyset$
$trv = w : w.id = max\{w'.id : w' \in W\}$
$val \leftarrow w.val$
elsif $R \neq \emptyset$
$trv = r : r.id = max\{r'.id : r' \in R\}$
$trv.batch \leftarrow queue \setminus \{trv\}$
$trv.phase \leftarrow \mathsf{starting}$
$trv.intr \leftarrow i$
$trv.dirs \leftarrow \{E\}$
$treating \leftarrow treating \cup \{trv\}$
$queue \leftarrow \emptyset$
$treat\text{-}time \leftarrow clock + treat\text{-}fqcy$

Internal $\mathsf{trv\text{-}prop}(trv)_i$
Precondition:
$\neg failed \wedge replica$
$trv \in treating$
$trv.type = \mathsf{write}$
$acc[trv.tid] = \{E\}$
Effect:
$W = \{w \in trv.batch : w.type = \mathsf{write}\}$
$trv.tag \leftarrow \langle trv.tag.ct + 1, i, |W|\rangle$
$trv.dirs \leftarrow \{N, S\}$
$acc[trv.tid] \leftarrow \emptyset$
$has\text{-}changed \leftarrow \mathsf{true}$

Internal $\mathsf{set\text{-}lock}(tid)_i$
Precondition:
$\neg failed \wedge replica$
$trv.tid = tid$
$trv.phase = \mathsf{starting}$
$trv.type = \mathsf{read}$
$pending\text{-}props \neq \emptyset$
Effect:
$\forall \langle p, *\rangle \in pending\text{-}props$
$locked[tid] \leftarrow locked[tid] \cup \{p\}$
$trv.phase = \mathsf{waiting}$

Output $\mathsf{trv\text{-}involve}(m)_i$
Precondition:
$\neg failed \wedge replica$
$trv \in treating$
$m.tid = trv.tid$
$m.type = trv.type$
$m.intr = trv.intr$
$m.dir \in trv.dirs$
$m.tag \in trv.tag$
$m.val \in trv.val$
$locked[m.tid] = \emptyset$
Effect:
none

Input $\mathsf{trv\text{-}participate}(m)_{j,i}$
Effect:
if $\neg failed \wedge replica$
$trv \leftarrow \mathsf{get\text{-}trv}(treating, m.tid)$
if $trv = \perp$
$trv.tid \leftarrow m.tid$
$trv.type \leftarrow m.type$
$trv.intr \leftarrow m.intr$
$trv.tag \leftarrow m.tag$
$trv.val \leftarrow m.val$
$treating \leftarrow treating \cup \{trv\}$
if $trv.intr = i$
$acc[trv.tid] \leftarrow acc[trv.tid] \cup \{m.dir\}$
else
$trv.dirs \leftarrow trv.dirs \cup \{m.dir\}$
$treating \leftarrow treating \cup \{trv\}$
if $type = \mathsf{write} \wedge m.dir \in \{N, S\}$
if $\langle m.tid, *\rangle \in pending\text{-}props$
$\forall tid', locked[tid'] \leftarrow locked[tid'] \setminus \{m.tid\}$
else
$pending\text{-}props \leftarrow pending\text{-}props \cup \{\langle m.tid, m.dir\rangle\}$
if $tag < trv.tag$
$\langle tag, val\rangle \leftarrow \langle trv.tag, trv.val\rangle$
else
$\langle trv.tag, trv.val\rangle \leftarrow \langle tag, val\rangle$
$has\text{-}changed \leftarrow \mathsf{true}$

Input $\mathsf{update\text{-}adj\text{-}om}(t, v, pp)_i$
Effect:
if $\neg failed \wedge replica$
$tag \leftarrow t$
$val \leftarrow v$
$pending\text{-}props \leftarrow pp$

Output $\mathsf{update\text{-}om\text{-}adj}(t, v, pp)_i$
Precondition:
$\neg failed \wedge replica$
$has\text{-}changed$
$\langle t, v\rangle = \langle tag, val\rangle$
$pp = pending\text{-}props$
Effect:
$has\text{-}changed \leftarrow \mathsf{false}$

Figure 5: $OperationManager_i$ automaton: Traversal transitions

needs to be expanded (no overloaded replica has been found), the load balancer automaton activates the adjuster automaton to add another replica to the quorum system. Finally, if for a certain amount of time no write or read

request have been locally submitted, then the load balancer automaton invokes the adjuster automaton for a shrink procedure: the local replica is removed from the quorum system.

4.2.1 States

The state variable used for by the $LoadBalancer$ are described in Figure 4. Each node maintains a local fixed-size $queue$ where read/write requests, i.e. traversals, are enqueued before treatment.

As mentioned in Section 4.1.1, boolean $failed$ field indicates if the replica is active or not. Dequeued operations are performed periodically. Variable $batch$ represents the batch of requests dequeued periodically, $acked$ the set of requests to acknowledge. Variable $starter$ is the identifier of the node that starts the thwart process. The $zone$ record contains the limits of the responsibility zone. Variable fwd is the node identifier to whom the thwart message is forwarded.

4.2.2 Transitions

Transitions of $LoadBalancer$ are defined in Figure 6. An operation event (that is a read or write operation) is triggered by a client. This event creates a new trv with a unique identifier and adds it to the $queue$ of the replica. Regarding to the $treat\text{-}time$ and $treat\text{-}fqcy$ fields and the replica local $clock$, trv-cons$_i$ event is triggered periodically. The queue $queue$ is scanned: the last enqueued write request (if any) is chosen for a write traversal. If there is no write requests, then one of the read requests is chosen for a read traversal. Let trv be the chosen request. Since a write traversal contains a consultation phase, there is no need to trigger a read traversal whenever a write traversal is triggered. The subfield $trv.batch$ is filled with other traversal from the $queue$ to keep track of all the clients to which the status/result of their operation will be sent (after completion of trv traversal). Queue $queue$ is emptied and $treat\text{-}time$ is reset. Then the chosen traversal trv is executed like explained in 4.1.2, while the batched ones are overlapped: they are not explicitly run but their result depend on their representative one, trv. Completion of the traversal is indicated upon receipt of a operation-ack$_{i,*}$ event. The traversal is assigned to value v' and its phase is set to ending. An operation acknowledgment is sent through operation-ack$_{i,*}$ for all batched traversals associated with traversal trv.

If clients requests are too frequent with respect to the $timeout$ value, the $queue$ might get full. In this case, a thwart$_i$ event occurs and the thwart process starts. That is, a traversal request is dequeued and prepared to be forwarded. See Section 4.2.3, for more details on how the next thwart replica is chosen. The requests is forwarded by the $Adjuster_i$ automaton that finds the correct thwart neighbor, according to the direction mentioned by the $OperationManager_i$ and conveyed by a $thwart\text{-}involve_i$ output event. As soon as $Adjuster_j$ receives the request it sends it to its $OperationManager_j$ automaton with a $thwart\text{-}partipate_j$ event to make it participate in turn. The thwarter process works as follows: either a non overloaded replica is found, in which case this replica becomes the initiating replica for the request, or the $thwart$ message has completed the diagonal path (i.e., it is received by replica $starter$). In this case, $starter$ decides to expand the quorum system by executing an expand$_i$ output event contacting the $Adjuster_i$ automaton. The traversal is kept in a freezing state until the sufficiently neighbors have acknowledge the end of the expand procedure by the mean of an heartbeat message.

It may happen that the queue of a node remains empty for most of the time: this may indicate that the size of the quorum is too big with respect to the actual load. The $LoadBalancer_i$ decides in this case to remove the process itself from the quorum, by running the output shrink$_i$ event, which subsequently provokes a corresponding input event in the $Adjuster_i$ automaton. The shrink event is triggered if upon expiration of a timeout, defined by $shrink\text{-}time$, $shrink\text{-}fqcy$ and the $clock$, the queue is empty. Please note that more sophisticated triggering policies can be introduced to avoid the system to continuously bounce between expansion and shrink. These aspects, as well as correct setting of timeouts are out of the scope of this report and will be investigated in future work.

4.2.3 The Thwart Path

The thwarter mechanism aims at selecting nodes and test if their load allows them to initiate the traversal. If not, the search is propagated following a *thwart path*. Let i be the *starter* of the thwart procedure. Roughly, the thwart path follows a diagonal direction from the starter. Figure 7 shows an example of overlay with a thwarter path starting from replica whose responsibility is the light gray zone. When a replica j, different from the starting node, decides to forward a thwart message, it sends it to a neighbor in the direction of a trajectory d' which starts from starter and is parallel to the diagonal d of the overlay square. d' is indicated with a light dashed line in Figure 7. The thwart message

Input $\mathsf{operation}(type, v)_{j,i}$
Effect:
  if $\neg failed \wedge replica$
    if $type = \mathsf{read}$
      $r\text{-}access \leftarrow r\text{-}access + 1$
    else
      $w\text{-}access \leftarrow w\text{-}access + 1$
    $ct \leftarrow ct + 1$
    $trv \leftarrow \langle\langle ct, i\rangle, type, j, i, v, \bot, \mathsf{idle}, \emptyset, \bot\rangle$
    if $|queue| \geq threshold$
      $fwd \leftarrow fwd \cup trv$
      $starter \leftarrow i$
    else
      $queue \leftarrow queue \cup \{trv\}$
    $shrink\text{-}time \leftarrow clock + shrink\text{-}fqcy$
    $has\text{-}changed \leftarrow \mathsf{true}$

Internal thwart_i
Precondition:
  $\neg failed \wedge replica$
  $|queue| \geq threshold$
Effect:
  $fwd \leftarrow fwd \cup trv$:
    $trv.tid = max\{trv'.tid : trv' \in queue\}$
  $queue \leftarrow queue \setminus \{fwd\}$
  $starter \leftarrow i$

Output $\mathsf{thwart\text{-}involve}(m)_i$
Precondition:
  $\neg failed \wedge replica$
  $fwd \neq \emptyset$
  $trv \in fwd$
  $m.tid \leftarrow trv.tid$
  $m.type \leftarrow trv.type$
  $m.str = starter$
Effect:
  none

Input $\mathsf{thwart\text{-}participate}(m)_{j,i}$
Effect:
  if $\neg failed \wedge replica$
    if $(starter = i)$
      $order\text{-}expand \leftarrow \mathsf{true}$
    else
      $trv.tid \leftarrow m.tid$
      $trv.type \leftarrow m.type$
      if $|queue| \geq threshold$
        $fwd \leftarrow fwd \cup trv$
        $starter \leftarrow m.str$
      else
        $queue \leftarrow queue \cup \{trv\}$

Output shrink_i
Precondition:
  $\neg failed \wedge replica$
  $clock \geq shrink\text{-}time$
  $queue = \emptyset$
  $replica = \mathsf{true}$
Effect:
  if $r\text{-}access = w\text{-}access = 0$
    $replica \leftarrow \mathsf{false}$
    $shrink\text{-}time \leftarrow \infty$
  else
    $r\text{-}access \leftarrow 0$
    $w\text{-}access \leftarrow 0$

Output $\mathsf{expand}(ra, wa)_i$
Precondition:
  $\neg failed \wedge replica$
  $order\text{-}expand = \mathsf{true}$
  $ra = r\text{-}access$
  $wa = w\text{-}access$
Effect:
  $order\text{-}expand \leftarrow \mathsf{false}$

Output $\mathsf{operation\text{-}ack}(v)_{i,j}$
Precondition:
  $\neg failed \wedge replica$
  $trv \in treating$
  $t \in trv.batch$
  $j = t.rqstr$
  $v = trv.val$
  $acc[trv.tid] = \{N, S\} \wedge trv.type = \mathsf{write}$
  $acc[trv.tid] = \{E\} \wedge trv.type = \mathsf{read}$
Effect:
  $trv.batch \leftarrow trv.batch \setminus \{t\}$

Output $\mathsf{operation\text{-}ack}(v)_{i,j}$
Precondition:
  $\neg failed \wedge replica$
  $trv \in treating$
  $trv.batch = \emptyset$
  $j = trv.rqstr$
Effect:
  $trv.phase \leftarrow \mathsf{ending}$
  $treating \leftarrow treating \setminus \{trv\}$

Figure 6: $OperationManager_i$ automaton: Load balancer transitions

is forwarded to the neighbor corresponding to the edge that intersects the diagonal (or the closest neighbor in the north direction). Since every replica knows its own zone limits and the bounds of the coordinate space, we only need to propagate the starting replica coordinates, to define a diagonal routing strategy that would eventually reach back the starting replica. The thwart message checks all the replicas whose zone intersects the diagonal and this guarantees that the initial zone is eventually reached again if every overloaded replica forwards this message. However, it may happen that the starter fails while the thwart is in progress, or that a new replica is inserted within the starter zone. In order to guarantee the thwart termination, the starter replica initially indicates to all its neighbors that a thwart process

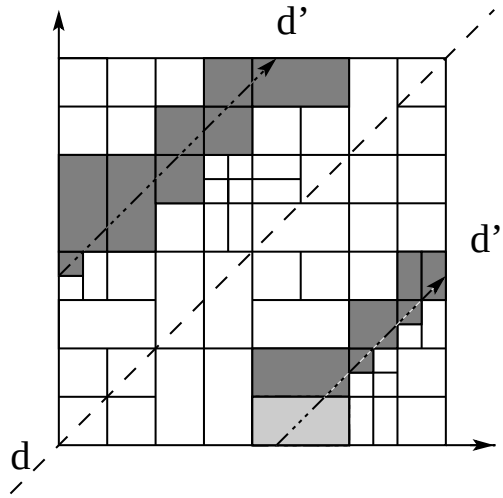

Figure 7: A Thwart Path

is started. It does the same in case of expansion. The replica that takes over the starter's zone is aware of the thwart process and stop it if it receives the thwart message.

For the sake of clarity we do not add these details in the thwarter code.

4.3 Adjuster

The adjuster automaton manages the structure of the overlay. In particular, it handles expansion of the quorum system and sent and receipt of message, either subsequent to an environmental event or requested by the $LoadBalancer$, and departures of existing replicas, either due to voluntary leaves or failures.

4.3.1 States

States of automaton $Adjuster$ are presented in Figure 8. The $zone$ of a replica is a rectangle that has four reals $x1, x2, y1, y2$ defining its limits in the two-dimensional coordinate space, $x1$ and $x2$ are respectively the minimum and the maximum abscissae whereas $y1$ and $y2$ are respectively the minimum and the maximum ordinates. The nbr field gives information about a neighbor, its $zone$, its id, its neighbors identity nid and the running propagation phases pp it knows about. Notice that the tag and val fields of the $OperationManager$ are also present, here. Each node knows about the fixed $bounds$ of the coordinate space. The *freezing* field prevents from sending messages during an expansion. The $involving\text{-}msg$ field is used to record the messages that involve another replica (i.e., messages that must be sent to its $Adjuster$ automaton). And the $participating\text{-}msg$ records messages that have to make the current replica participate (i.e., messages that must be sent to the $OperationManager$ automaton).

4.3.2 Transitions

The self-healing and self-adjusting behaviors come from the $Adjuster$ automaton, whose local transitions appear in Figure 9 and other transitions appear in Figure 10. There are roughly three important services provided by this automaton: (i) it provides message-passing communication, (ii) it reduces the set of active replicas responsible of the atomic memory by forcing a replica to exit the system, and (iii) it expands this set by choosing a replica candidate.

By the expand_i and the shrink_i events, the system self-adjusts. The expand_i event is triggered by the $LoadBalancer_i$ if the load becomes sufficiently high, while the shrink_i event is triggered by the $LoadBalancer_i$ if no request has been received by i during a sufficiently long period. Some heartbeat messages are preriodically exchanged between $Adjuster$ automaton of different replicas in order to exchange neighbors current state. Likewise, expand messages are exchanged between a replica and the additional node it choose. More generally, messages of each type are exchanged by the mean of $\mathsf{send}(\mathsf{type}, \ldots)$ and $\mathsf{recv}(\mathsf{type}, \ldots)$ actions. Finally, the update-adj-om and update-om-adj actions allow $OperationManager$ and $Adjuster$ to update some freshly modified states they share (e.g., tag and val).

States:

$zone$ a record with fields
- $x_1 \in \mathbb{R}$
- $x_2 \in \mathbb{R}$
- $y_1 \in \mathbb{R}$
- $y_2 \in \mathbb{R}$

nbr a record with fields
- $zone \in \mathbb{R}^4$
- $id \in I$
- $nid \subset I$
- $pp \in TId \times \{N, S, E\}$

m a record with fields
- $tid \in TId$
- $type \in \{\mathsf{read}, \mathsf{write}\}$
- $intr \in I$
- $str \in I$
- $dir \in \{N, S, E\}$
- $line \in \mathbb{R} \times \mathbb{R}$

tag a record with fields
- $ct \in \mathbb{N}$
- $id \in I$
- $index \in I$

$val \in V$
$bounds$ a zone
$nbrs$ an array of neighbor nbr
$failed$ a boolean
$has\text{-}changed$ a boolean
$involving\text{-}msg$ a set of messages
$participating\text{-}msg$ a set of messages
$replica$ a boolean
$clock \in \mathbb{R}^{>0}$
$hb\text{-}time \in \mathbb{R}^{>0}$
$hb\text{-}fqcy \in \mathbb{R}^{>0}$
$last\text{-}split \in I$
$a\text{-}last\text{-}split$ a mapping from I to a boolean
$detect\text{-}time$ a mapping form I to $\mathbb{R}$
$freezing \subseteq \{N, S, E\}$
$coeff \in \mathbb{R}^{>0}$

Signature:

Input:
- $\mathsf{trv\text{-}involve}(m)_i,\ i \in I$
- $\mathsf{thwart\text{-}involve}(m)_i,\ i \in I$
- $\mathsf{shrink}_i,\ i \in I$
- $\mathsf{expand}(ra, wa)_i,\ i \in I, ra, wa \in \mathbb{N}$
- $\mathsf{update\text{-}om\text{-}adj}(t, v, pp)_i,\ i \in I,\ t \in T,\ v \in V,\ pp \in TId \times \{N, S, E\}$
- $\mathsf{recv}(*)_{j,i},\ i \in I$

Internal:
- $\mathsf{heal}_i,\ i \in I$

Output:
- $\mathsf{trv\text{-}participate}(m)_i,\ i \in I$
- $\mathsf{thwart\text{-}participate}(m)_i,\ i \in I$
- $\mathsf{update\text{-}adj\text{-}om}(t, v, pp)_i,\ i \in I,\ t \in T,\ v \in V,\ pp \in TId \times \{N, S, E\}$
- $\mathsf{send}(*)_{i,j},\ i \in I$

Figure 8: $Adjuster_i$ automaton: Signature and States

Communication The $Adjuster$ acts as a communication medium between higher level $OperationManager$ automata (see Figure 1). That is when messages need to be exchanged between $Traversals$ or between $LoadBalancers$ for the thwart mechanism, the $OperationManager_i$ outputs corresponding events $\mathsf{trv\text{-}involve}_i$ or $\mathsf{thwart\text{-}involve}_i$ to the $Adjuster_i$ with enough information. Specifically, this information informs the $Adjuster_i$ automaton if the message is part of a thwart or a traversal and in what direction. Since this automaton maintains information on i's zone and its neighbors zone coordinates, it can find the right neighbor j among all to contact. That is, $Adjuster_i$ sends $\mathsf{operation}$ messages to another $Adjuster_j$ automaton. When $Adjuster_j$ receives this message it conveys it immediately to its $OperationManager_j$ automaton by the mean of the $\mathsf{trv\text{-}participate}_j$ or $\mathsf{thwart\text{-}participate}_j$ output event.

Expansion More precisely, in some cases the $LoadBalancer_i$ does not execute the traversal directly. This case occurs when replica i is already overloaded and the thwart strategy informs about a high loaded system. That is, replica i decides to add a replica in the quorum system. Since i knows that no replica wants to run the traversal, it triggers the expansion process at the $Adjuster_i$ by an $expand$ synchronized event. This event tells the $Adjuster_i$ about the concerned traversal trv. Then a node, referred as $to\text{-}add$ in the algorithm, is chosen arbitrarily among a non empty set of candidates. Observe that those candidates may not own a replica of the data and are not considered as part of the memory at this time. Next, i splits its zone in two halves. The splitting can be done either vertically or horizontally. Node i force replication to $to\text{-}add$, keeps the responsibility of one of the two zones and gives the other one to the new replica $to\text{-}add$. The split direction depends on read/write access histories (ra and wa): assuming that a write traversal has to probe $coeff$ times more replicas than a read traversal, i aims to minimize (by its split choice) the average probe-complexity of further traversals. Because of the split zone, the $to\text{-}add$ becomes neighbor of i, thus i adds it in its list of neighbors $nbrs$ and considers itself as $a\text{-}last\text{-}split[to\text{-}add]$, i.e. the one who shared its zone with this entering replica.

Input trv-involve$(m)_i$
Effect:
  if $\neg failed \wedge replica$
    if $m.intr = i$
      $m.line \leftarrow$ get-straight-line$(zone, m.dir)$
    $m.next \leftarrow$ get-nbr$(nbrs, m.line)$
    $involving\text{-}msg \leftarrow involving\text{-}msg \cup \{m\}$
    if $m.type =$ write
      $propagate\text{-}line[m.tid] \leftarrow m.line$

Output trv-participate$(m)_i$
Precondition:
  $\neg failed \wedge replica$
  $m \in participating\text{-}msg$
  $m.dir \neq \perp$
Effect:
  $participating\text{-}msg \leftarrow participating\text{-}msg \setminus \{m\}$

Input shrink$_i$
Precondition:
  $\neg failed \wedge replica$
  $last\text{-}split = \perp$
Effect:
  $replica \leftarrow$ false

Input expand$(ra, wa)_i$
Effect:
  if $\neg failed \wedge replica$
    $to\text{-}add \leftarrow$ get-outside-node()
    $replica \leftarrow$ true
    if $((coeff \times ra) > wa)$
      $to\text{-}add.zone \leftarrow \langle y_1 + (y_2 \perp y_1)/2, y_2, x_1, x_2 \rangle$
      $zone \leftarrow \langle y_1, y_1 + (y_2 \perp y_1)/2, x_1, x_2 \rangle$
      $freezing \leftarrow freezing \cup \{N\}$
    else
      $to\text{-}add.zone \leftarrow \langle y_1, y_2, x_1 + (x_2 \perp x_1)/2, x_2 \rangle$
      $zone \leftarrow \langle y_1, y_2, x_1, x_1 + (x_2 \perp x_1)/2 \rangle$
      $freezing \leftarrow freezing \cup \{E\}$
    $index \leftarrow |nbrs| + 1$
    $nbrs[index] \leftarrow to\text{-}add$
    $a\text{-}last\text{-}split[to\text{-}add] \leftarrow$ true

Input thwart-involve$(m)_i$
Effect:
  if $\neg failed \wedge replica$
    if $m.str = i$
      $m.line \leftarrow$ get-diagonal$(zone, bounds)$
    $m.next \leftarrow$ get-nbr$(nbrs, m.line)$
    $involving\text{-}msg \leftarrow involving\text{-}msg \cup \{m\}$

Output thwart-participate$(m)_i$
Precondition:
  $\neg failed \wedge replica$
  $m \in participating\text{-}msg$
  $m.dir = \perp$
Effect:
  $participating\text{-}msg \leftarrow participating\text{-}msg \setminus \{m\}$

Output update-adj-om$(t, v, pp)_i$
Precondition:
  $\neg failed \wedge replica$
  $has\text{-}changed$
  $\langle t, v \rangle = \langle tag, val \rangle$
  $pp = pending\text{-}props$
Effect:
  $has\text{-}changed \leftarrow$ false

Input update-om-adj$(t, v, pp)_i$
Effect:
  if $\neg failed \wedge replica$
    $tag \leftarrow t$
    $val \leftarrow v$
    $pending\text{-}props \leftarrow pp$

Figure 9: $Adjuster_i$ automaton: Transitions

Shrink The internal shrink action is triggered internally by the $Adjuster_i$ automaton. It may happen that a replica of any quorum system have not received any request since a long time. In this case, we reasonably assume that a random access strategy implies that the quorum system load is low. That is, if a replica i has not received any request during a *shrink-fqcy* period of time, the replica takes the decision to shrink the system by removing one replica. Hence the $OperationManager_i$ outputs a shrink$_i$ event to the $Adjuster_i$ automaton. When the $Adjuster_i$ and the $OperationManager_i$ automata run such an event, the *replica* field of node i is set to false. Consequently node i can not runs any further action. By doing so, the system shrink process is comparable to a crash failure where the crashed node do not notify before failing. However this scheme is easily changeable to become a graceful leave, where the replica having split for the last time is chosen among neighbors of i, and i's zone responsibility is given back to the corresponding chosen replica. Since we already assumed the use of CAN takeover mechanism without specifying it, we simply mention that this shrink procedure is easily changeable into a notified leave leading more rapidly than a traditional failure to the same takeover result.

Output $\mathsf{send}(\langle \mathsf{heartbeat}, pp, pl, z, nid, ls, t, v\rangle)_{i,j}$
Precondition:
$\neg failed \wedge replica$
$j \in nbrs$
$hb\text{-}time \geq clock$
$pp = pending\text{-}props$
$pl = propagate\text{-}line$
$z \leftarrow zone$
$nid \leftarrow \bigcup_{n \in nbrs} \{n.id\}$
$ls \leftarrow last\text{-}split$
$t \leftarrow tag$
$v \leftarrow val$
Effect:
$hb\text{-}time = clock + hb\text{-}fqcy$

Input $\mathsf{recv}(\langle \mathsf{heartbeat}, pp, pl, z, nid, ls, t, v\rangle)_{j,i}$
Effect:
if $\neg failed \wedge replica$
if $\forall index : nbrs[index].id \neq j$
$index \leftarrow |nbrs| + 1$
$nbrs[index].id \leftarrow j$
else
let $index$ be st. $nbrs[index].id = j$
$nbrs[index].zone \leftarrow z$
$nbrs[index].nbrs \leftarrow nid$
$nbrs[index].pp \leftarrow pp$
$nbrs[index].pl \leftarrow pl$
$detect\text{-}time[j] \leftarrow clock + detect\text{-}fqcy$
if $ls \neq i$
$a\text{-}last\text{-}split[j] \leftarrow \mathsf{false}$
if $\bigcup_{j' \in nbrs} \{[j'.zone.y_1, j'.zone.y_2] :$
$j'.zone.x_1 > zone.x_1\} \subset [zone.y_1, zone.y_2]$
$freezing \leftarrow freezing \setminus \{E\}$
if $\bigcup_{j' \in nbrs} \{[j'.zone.x_1, j'.zone.x_2] :$
$j'.zone.y_1 > zone.y_1\} \subset [zone.x_1, zone.x_2]$
$freezing \leftarrow freezing \setminus \{N\}$
if $\bigcup_{j' \in nbrs} \{[j'.zone.x_1, j'.zone.x_2] :$
$j'.zone.y_1 < zone.y_1\} \subset [zone.x_1, zone.x_2]$
$freezing \leftarrow freezing \setminus \{S\}$
if $t > tag$
$\langle tag, val\rangle \leftarrow \langle t, v\rangle$
$has\text{-}changed \leftarrow \mathsf{true}$
if $freezing = \emptyset$
$pending\text{-}props \leftarrow \mathsf{update\text{-}pp}(nbrs, pl)$
$has\text{-}changed = \mathsf{true}$

Output $\mathsf{send}(\langle \mathsf{expand}, v, t, neighbors\rangle)_{i,j}$
Precondition:
$\neg failed \wedge replica$
$j = to\text{-}add$
$t = tag$
$v = val$
$neighbors = nbrs$
Effect:
none

Input $\mathsf{recv}(\langle \mathsf{expand}, v, t, neighbors\rangle)_{j,i}$
Effect:
if $\neg failed \wedge \neg replica$
$val \leftarrow v$
$tag \leftarrow t$
$has\text{-}changed \leftarrow \mathsf{true}$
$nbrs \leftarrow \mathsf{update\text{-}nbr}(zone, neighbors)$
$last\text{-}split \leftarrow j$
$replica \leftarrow \mathsf{true}$
$freezing \leftarrow \{N, S, E\}$

Output $\mathsf{send}(\langle \mathsf{operation}, m\rangle)_{i,j}$
Precondition:
$\neg failed \wedge replica$
$m \in involving\text{-}msg$
$j = m.next$
$freezing = \emptyset$
Effect:
none

Input $\mathsf{recv}(\langle \mathsf{operation}, m\rangle)_{j,i}$
Effect:
if $\neg failed \wedge replica$
$participating\text{-}msg \leftarrow participating\text{-}msg \cup \{m\}$

Figure 10: $Adjuster_i$ automaton: Transitions

5 SAM Atomicity

In this section we prove that our system implements atomic objects. Hence, we show that linearizability of operations is ensured despite node arrivals, departures and single phase read operation.

5.1 Assumptions and Preliminary Definitions

5.1.1 Quorums Properties

The read operation consists in contacting each node of a consultation quorum while the write operation contacts first a consultation quorum and then a propagation quorum. By theorem 3.1, each type of quorum intersects any quorum of the second type. Any failure makes the traversal waiting until a replica takes over the state of the failing node. If

a replica crashes, it involves the crash of the quorum it belongs to. As this time since no traversal is possible through one of this quorum, concerned traversals remain in a waiting state until the overlay is healed.

In the remaining of the report we use traversal terminology to describe the strategy used during an operation. That is a read (resp. write) traversal refers to a read (resp. write) operation. Notice that there is a bijective mapping from each operation to each traversal since operation event sets a new traversal identifier each time it occurs.

5.1.2 Tag to Traversal assignment

Definition 4 (Batched Traversal) *A treated traversal trv maintains a set, possibly empty, the $trv.batch$ subfield. This set is filled out, when a* trv-cons *event occurs, with all the other traversals of the queue. Those traversals are called* batched traversals *and traversal trv is called their* representative.

We propose a tag to traversal assignment such that each treated traversal gets assigned a monotonically incremented counter coupled with a tie-breaker identifier. We add a batch index to this tag in order to differentiate traversals that are overlapped. For instance, the read batched traversals have the same tag as their representative while the write batched traversals get assigned a tag lower than their representative's and higher than any lower representative tag. This later tag is one of the immediate preceding tags of their representative one.

We refer to **tag** as a mapping from TId to $\mathbb{N} \times I \times \mathbb{N}$ such that $trv.tid$ is mapped to $tag(trv.tid)$ if at least one of the following condition holds: (i) if $trv.type = \mathsf{read}$, then $tag(trv.tid) = trv'.tag_i$ and $trv \in trv'.batch_i$ when $\mathsf{operation\text{-}ack}(trv', v)_i$ occurs, where $trv'.tag.id = i$. (ii) $trv.type = \mathsf{write}$ and if $trv \in trv'.batch$ then $tag(trv.tid) = \langle trv'.tag.ct{+}1, i, index\rangle$ with $index$ the index of trv in the ordered set $trv'.batch$ when $\mathsf{trv\text{-}prop}(trv')_i$ occurs. (iii) $\nexists trv'$ such that $trv \in trv'.batch$ and if $trv.type = \mathsf{read}$ then $tag(trv.tid) = tag_i$ when $\mathsf{operation\text{-}ack}_i$ occurs, else $tag(trv.tid) = tag_i$ immediately after $\mathsf{trv\text{-}prop}_i$ occurs.

We define the ordering of tag as follow:

- $tag_1 < tag_2$ if and only if
  - $tag_1.ct < tag_2.ct$ or
  - $tag_1.ct = tag_2.ct$ and $tag_1.id < tag_2.id$ or
  - $tag_1.ct = tag_2.ct$ and $tag_1.id = tag_2.id$ and $tag_1.b\text{-}ind < tag_2.b\text{-}ind$
- $tag_1 = tag_2$ if and only if neither $tag_1 < tag_2$ nor $tag_2 < tag_1$

5.1.3 Real-Time Precedence.

We refer to **start** as a mapping from a traversal to $\mathbb{R}^+$ such that traversal tid is mapped to time τ if $\mathsf{operation}(*, *)$ occurs at time τ with $tid = trv.tid$. We define **term** as a mapping from a traversal or an operation to $\mathbb{R}^+$ such that traversal tid (resp. operation π) is mapped to time τ' if $\mathsf{trv\text{-}ack}(tid, *)$) (resp. the corresponding $\mathsf{operation\text{-}ack}(*)$) event occurs at time τ' with if $trv.batch \neq \emptyset$ then $tid = t.tid$ else $tid = trv.tid$.

Definition 5 (History's precedence) *First, let $\prec$ be a total order capturing the* real-time *precedence on every event. Second, we define the history precedence, namely $<_H$, as an irreflexive partial order between operations π_1 and π_2, such that $\pi_1 <_H \pi_2$ if and only if $term(\pi_1) \prec start(\pi_2)$.*

Definition 6 (Atomicity) *Next, we restate the Definition 1. If it exists a matching $\mathsf{operation\text{-}ack}_{x,i}$ event following any $\mathsf{operation}_{x,i}$ event then, it exists a relation $<_S$ that orders partially read and write traversals such that:*

1. $<_H \subseteq <_S$.
2. *For any write traversal w_1 and w_2, either $w_1 <_S w_2$ or $w_2 <_S w_1$.*
3. *For any read traversal r and write traversal w, either $r <_S w$ or $w <_S r$. Moreover, if it exists w such that $w = max_{<_S}\{w' <_S r\}$ and w writes value v then r returns the same value v. And if no such w exists, then the value returned by r is v_0.*
4. *For any read or write traversal τ, the set $\{\tau' : \tau' <_S \tau\}$ is finite.*

5.1.4 Assumptions

First, we assume the presence of the non specified functions of the $Adjuster$. Then we assume that execution sequences are well-formed and we use CAN takeover mechanism.

Additional functions

- **get-outside-node** Now we assume reasonably that if the quorum system is considered to be overloaded then the whole system contains more than the quorum system nodes. This is reasonable since the quorum system overload comes from an increasing total number of nodes in the system over the number of replicas. We assume that at least one node remains unused as a replica yet that is active in the system when an expand event occurs. We assume for the bootstrapping process that each active node can invoque get-outside-node() function which returns the identity of one node among those ones.
- **get-straight-line** Given the zone z of a replica and a direction, this function returns the line that should follow the corresponding traversal to wrap the torus. Typically it is the horizontal or vertical line going through the middle of zone z.
- **get-diagonal** Given the zone z of a replica and the general bounds, of the coordinates space, this function returns the line parallel to the diagonal of the space that crosses the middle of zone z
- **get-nbr** Given the set of neighbors, and consequently the zone limits of each of them, and a line define by two reals, namely a and b, this function returns the neighbor whose zone crossed the line L of equation $y = ax + b$. If such a replica does not exist, the default chosen one is the north neighbor with $y.xmax \in L$.
- **update-nbr** Given a set of neighbors including all the neighbors of a replica i, plus responsibility zone of i, this function returns the exact set of neighbors of i.
- **update-pp** Given the set of i's neighbors and consequently their pending propagation identifier and direction, the line followed by these traversals, this function returns the state of the current replica. That is, this replica knows if it has to continue a pending traversal, if it is the leader and if it is locked.

Well-formedness. We assume that any sequence of external actions for replica i and object x is *well-formed*. That is (i) the first event of the sequence is either an $\mathsf{operation}_{*,i}$ event or a fail_i event. (ii) an $\mathsf{operation}_{*,i}$ event is immediately followed by the matching $\mathsf{operation\text{-}ack}_{i,*}$. (iii) no fail_i event precedes any $\mathsf{operation}_{*,i}$, or $\mathsf{operation\text{-}ack}_{i,*}$ event.

Takeover mechanism. We do not include the details of CAN takeover mechanism. We rather assume that failure rate is low enough to ensure the takeover completes. While operations can be stopped during this mechanism, they terminate when quorums are newly available.

5.2 Atomicity Proof.

Here, we aim at showing that it exists a relation $<_t$ defined by operation tag, that is a partial order satisfying atomicity.

Definition 7 (Traversal ordering) *We define a partial order $<_t$ on the set of traversals such that (i) write traversals are totally ordered: a write traversal π_1 precedes another write traversal π_2 if $tag(\pi_1) < tag(\pi_2)$. (ii) read traversals are ordered between write traversals: a read traversal π_1 is ordered after all write traversals π_ℓ such that $tag(\pi_\ell) \leq tag(\pi_1)$ and before all write traversals $\pi_{\ell'}$ such that $tag(\pi_1) < tag(\pi_{\ell'})$.*

The first invariant shows that any locked node can not participate in any read traversal until it is unlocked.

Invariant 1 *If a replica j is locked before a read traversal t consults it, then traversal t and all its possible batched traversal terminate after the replica is unlocked.*

Proof. Assume that j is locked when the consultation phase of t consults it, i.e. when a $trv\text{-}participate(m)_j$ such that $m.tid = t$ occurs, it exists previous states where some $trv\text{-}participate(m')_j$ events occur as part of propagations of a set of write traversals identified by identifiers $t' = \{m'.tid\}$, namely T'. This former event fills out the $pending\text{-}props_j$ field with new traversal identifier and sense pair. It follows that $set\text{-}locks(t)_j$ preconditions are satisfied and $locked_j$ is filled with the traversal identifiers freshly added to $pending\text{-}props_j$. In this state, no further $trv\text{-}involve_j$ is possible, meaning that j can not involve its following neighbor in traversal t. The propagation traversals identified by T' terminates after a second $trv\text{-}involve(m'')$ output event occurs with $m''.tid \in T'$. By the $Adjuster$ automaton code, the corresponding input event record this message m'', trigging some $send(\langle \mathsf{operation}, m'' \rangle)_{*,j}$ output events. That is, $Adjuster_j$ receives the corresponding message and makes its $OperationManager_j$ participate with some $trv\text{-}participate_j$ event. From this point on, the corresponding write traversal identifiers are removed from $pending\text{-}prop$ indicating that the propagation is no more pending, and the $lock_j$ field is immediately set to $\emptyset$. Note that traversal identifiers whose propagation occurs after the $set\text{-}lock_j$ event do not lock traversal t. Since the termination of traversal t requires that the initiator gets recontacted by the same traversal t, this occurs at least after unlocking j. More over since any $\mathsf{operation\text{-}ack}(t, *)$ ends the batched traversal and occurs after the unlock of j, every batched traversal whose t is representative ends after j is unlocked. □

Next invariant states that any node of a propagation quorum is locked during a propagation until all the corresponding quorum's nodes have been locked.

Invariant 2 *Any write traversal t propagating to quorum q, unlocks any replica of q after all its replicas have been locked.*

Proof. Because of the torus topology we use, each consultation quorum as well as each propagation quorum is a ring. When one of the replicas of a quorum starts a propagation, it contacts its two neighbors in its propagation quorum. Like aforementioned, replicas learn about the traversal by a $trv\text{-}participate(m)$ where $m.tid = tid$ event and continues it by a $trv\text{-}involve(m)$ event, with $m.tid = tid$. By examination of the $trv\text{-}participate$ action, if $m.type = \mathsf{write}$ either $tid \in pending\text{-}props$ field and this identifier is added to $pending\text{-}prop$, or it is removed. In both cases, the $locked$ field is updated with new concurrent propagation phase discovered. Now observe that the propagation phase is done by the initiator i by sending messages in two opposite senses, namely N and S. Hence, when the first message is received by j, it goes to a state where it does not $\mathsf{involve}$ any more node in consultation traversals. However when j receives a second message, it goes back to an unlock state, being able to involve other nodes. The two messages are sent in both senses over the ring from a single node i. Because of the uniqueness of the path, each node receives one message locking it before one of them receives a second one, unlocking it. □

Here we aim at showing that the $\langle tag, val \rangle$ pair is up-to-date after an expansion. That is when a replica is added to the quorum system, a consultation or a propagation quorum changes. We show that this node keeps track of any earlier traversal, in other words its state reflects the traversals occurred before, when it becomes able to participate as a replica of a quorum.

Lemma 5.1 *After an expand_i event occurs including node j in the quorum system, at the time j becomes an active replica, its $\langle tag, val \rangle$ pair and $locked$ value reflects all the traversal participations whose quorums contain j.*

Proof. Preconditions of the $\mathsf{send}(\langle \mathsf{operation}, ... \rangle)_j$ prevent j from participating in a traversal before j becomes a replica (i.e. $replica_j = \mathsf{true}$) and $freezing_j$ is empty. Since $freezing_j$ is set to $\{N, S, E\}$ when j becomes a replica by $\mathsf{recv}(\langle \mathsf{expand}, ... \rangle)_j$, $\mathsf{heartbeat}$ messages have to be exchanged with its N, S and E neighbors before j can participate in another traversal. By the propagation phase of the write operation, vertical neighbors (i.e. N and S neighbors) of j have the updated $\langle tag, val \rangle$ pair of any quorum it belongs to. These later state message exchanges ensure that j updates its $\langle tag, val \rangle$ to the most up-to-date one when it starts participating in any further traversal. Observe finally that the pp is updated with the $\mathsf{update\text{-}pp}$ function like said in 5.1.4. □

The following lemma and corollary show that tag ordering respects real-time precedence ordering.

Lemma 5.2 *If $term(t_1) \prec start(t_2)$ then $tag(t_1) \leq tag(t_2)$.*

Proof. First observe by 5.1.2 that any batched traversal gets assigned a tag at most as large as their representative one. That is, we show the result true for t_1 a representative traversal and the result follows trivially for any of its batched traversals. We consider two cases, either t_1 is a read traversal or it is a write traversal.

In the first case assume that t_1 is a write traversal, hence t_1 completes only if its propagation phase has already ended. If $term(t_1) \prec start(t_2)$ that means that t_1 has propagated its tag to a whole propagation quorum, when traversal t_2 is initiated. By the quorum intersection property, it exists one element j that gets assigned this tag in every consulting quorum by time $term(t_1)$. Since tag value is monotonically incremented, and Lemma 5.1 and assumption ensure that any failure is followed by a state recovery, it is obvious that the tag tag' consulted during the consultation phase t_2 is such that $tag' \geq tag(t_1)$. By definition of $tag(t_2)$, it is such that $tag(t_2) \geq tag'$ and putting these inequalities together yields to the result.

Now consider the second case where t_1 is considered to be a read traversal. To prove that the property holds, we show that the following contraposition is true: If $tag(t_2) < tag(t_1)$ then $start(t_2) \prec term(t_1)$. Assume that t_1 consults the consulting quorum c_1 while t_2 consults the consulting quorum c_2. Hence if $tag(t_2) < tag(t_1)$ then $\exists j \in c_1$ such that $j.tag > max_{i \in c_2}\{i.tag\}$. Given that, we show that $term(t_1) \prec start(t_2)$ is impossible. The existence of j implies that it exists a write traversal t_3, propagating to a propagation quorum p_3, that has propagated to j but not yet to any replica of c_2. By the quorum intersection property, we know that $p_3 \cap c_2 \neq \emptyset$, then t_3 will eventually propagate to one element of c_2. By Invariant 2, j is locked until after having propagated to an element of c_2. Next, by Invariant 1, $term(t_1) \prec start(t_2)$ is impossible. That is, if $tag(t_2) < tag(t_1)$ then $start(t_2) \prec term(t_1)$ and the result is also true, i.e. if $term(t_1) \prec start(t_2)$ then $tag(t_1) \leq tag(t_2)$. □

Corollary 5.3 *If $term(t_1) \prec start(t_2)$ and t_2 is a write traversal then $tag(t_1) < tag(t_2)$.*

Proof. This follows directly from Lemma 5.2 and the definition of write traversal tag. Let tag_c be the tag at the end of consultation phase of t_2 before being incremented. By examination of $\mathsf{trv\text{-}prop}(t_2)$ action, we conclude that $tag(t_2) > tag_c$. Combining this with Lemma 5.2 leads to the conclusion. □

The main theorem shows that the traversal ordering defined in Definition 7, and based on tags, satisfies each one of the four conditions of the atomicity definition (Definitions 6 and 1).

Theorem 5.4 *SAM implements atomic object.*

Proof. Two write traversals get assigned different tags. This follows from the fact that two writes at the same location get assigned a different sequence number or a different low-order batched index, and writes occurring at different location get assigned different tie-breaker tag. That is Part 2 is satisfied. For Part 1, assume for the sake of contradiction that $<_H \not\subseteq <_t$. That is assume that $trv_1 <_H trv_2$ and $\neg(trv_1 <_t trv_2)$. Now there are two cases: (i) If trv_2 is a read traversal, then $trv_1 <_H trv_2$ and Lemma 5.2 implies that $tag(trv_1) \leq tag(trv_2)$. (ii) If trv_2 is a write traversal, then $trv_1 <_H trv_2$ and Corollary 5.3 implies that $tag(trv_1) < tag(trv_2)$. By definition of $<_t$, both results yield a contradiction. For Part 4, since any traversal trv_1 in H terminates, Lemma 5.2 implies that all traversals trv_2 such that $trv_1 <_H trv_2$ is ordered after. That is, the set of traversals preceding trv_1 is finite. Part 3 is straightforward. □

6 Conclusions and Future Work

In this report we have presented SAM, a system for emulating an atomic memory in highly dynamic systems. Our system has self-* capabilities, self-adjusting function of object load variations and self-healing when replicas leave the system. SAM follows a modular design inspired by theoretical and practical achievements in several research areas: p2p overlays, dynamic quorums and replica control. The philosophy of SAM is based on distributed control and locality principles. That is, all the good properties of SAM (atomicity, self-healing or self-adjusting) are implemented via p2p techniques using only local information whose size remains almost constant as the size of the sistem grows. Our work demonstrates that the use of p2p techniques can be beneficial for the future design of applications for dynamic systems with strong consistency requirements (eg. e-commerce, e-flows or e-booking).

Several research directions are opened by our work. First we intend to extend our approach to the implementation of a persistent storage in dynamic systems that support more complex operations. In [4] some of the authors have proposed an architecture for a persistent storage aimed to support multi-object operations. We decomposed the persistent storage problem in a set of sub-problems and each sub-problem was further decomposed in sub-problems that could be solved with classical distributed computing techniques and abstractions (called oracles) which implementation would need additional environmental assumptions. SAM is a fundamental abstraction (i.e. atomic memory for dynamic systems) aimed to be used as building block for implementing more complex services.

Another research direction would be to study incentive mechanisms for replication in dynamic systems and the impact of these mechanisms on the efficiency of the shared memory. Hosting a replica is resource consuming, hence nodes in the system may refuse to be part of an atomic memory. Solutions come from games and mechanisms theory which provide a broad class of incentive mechanisms.

Finally, we intend to study efficient mapping schemes between replicas that are part of the atomic memory and physical nodes in the network. SAM does not focus on the choice of a candidate for hosting a new replica. However, this choice may have a non-negligible impact on the latency and the availability of the atomic memory.

References

[1] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proc. of the 17th Int. Parallel and Distributed Processing Symposium (IPDPS)*, page 40, 2003.

[2] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. In Faith Ellen Fich, editor, *Distributed algorithms*, volume 2848/2003 of *Lecture Notes in Computer Science*, pages 60–74, 2003.

[3] D. Agrawal and A. El Abbadi. Efficient solution to the distributed mutual exclusion problem. In *Proc. of the 8th annual symposium on Principles of distributed computing (PODC)*, pages 193–200, 1989.

[4] E. Anceaume, R. Friedman, M. Gradinariu, and M. Roy. An architecture for dynamic scalable self-managed persistent objects. In *Proc. of Int. Symposium on Distributed Objects and Applications (DOA)*, pages 1445–1462, 2004.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[6] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Trans. Knowl. Data Eng.*, 4(6):582–592, 1992.

[7] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in ad hoc networks. In *Proc. of 17th International Symposium on Distributed Computing (DISC)*, pages 306–320, 2003.

[8] B. Englert and A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of of shared memory. In *Proc. of Int. Conf. on Distributed Computing Systems (ICDCS 2000)*, pages 454–463, 2000.

[9] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM symposium on Operating systems principles (SOSP'79)*, pages 150–162. ACM Press, 1979.

[10] S. Gilbert, N.A. Lynch, and A. Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of 17th Int. Conference on Dependable Systems and Networks (DSN)*, pages 259–269, 2003.

[11] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the 12th ACM symposium on Operating systems principles (SOSP'89)*, pages 202–210. ACM Press, 1989.

[12] M.P. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. on Database Systems*, 12(2):170–194, 1987.

[13] R. Holzman, Y. Marcus, and D. Peleg. Load balancing in quorum systems. *SIAM Journal on Discrete Mathematics*, 10(2):223–245, 1997.

[14] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. Computers*, 40(9):996–1004, 1991.

[15] L. Lamport. The part time parliament. *ACM Transactions on Computer Science*, 16(2):133–169, 1998.

[16] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, 2002.

[17] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report 3, CWI-Quaxterly, 1989.

[18] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[19] N.A. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorums. In *Proc. of 27th Int. Symp. on Fault-Tolerant Computing (FTCS)*, pages 272–281, 1997.

[20] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems (TOCS)*, 3(2):145–159, 1985.

[21] U. Nadav and M. Naor. Fault-tolerant storage in a dynamic environment. In *Proc. of he 18th Annual Conference on Distributed Computing (DISC)*, 2004.

[22] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[23] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[24] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. In *Proc. of the 22th annual symposium on Principles of distributed computing (PODC'03)*, pages 114–122. ACM Press, 2003.

[25] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM*, pages 161–172, 2001.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[27] B. Silaghi, P. Keleher, and B. Bhattacharjee. Multi-dimensional quorum sets for read-few write-many replica control protocols. In *In Proc. of the 4th CCGRID/GP2PC*, 2004.

[28] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. on Networking*, 11(1):17–32, 2003.