HAL
open science

Self-Adjusting Atomic Memory for Dynamic Systems based on Quorums On-The-Fly. Correctness Study

Emmanuelle Anceaume, Maria Gradinariu, Vincent Gramoli, Antonino Virgillito

To cite this version:

Emmanuelle Anceaume, Maria Gradinariu, Vincent Gramoli, Antonino Virgillito. Self-Adjusting Atomic Memory for Dynamic Systems based on Quorums On-The-Fly. Correctness Study. [Research Report] PI 1795, 2006, pp.15. inria-00001192

HAL Id: inria-00001192

https://inria.hal.science/inria-00001192v1

Submitted on 3 Apr 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.

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

PUBLICATION
INTERNE
N° 1795

SELF-ADJUSTING ATOMIC MEMORY FOR DYNAMIC SYSTEMS BASED ON QUORUMS ON-THE-FLY. CORRECTNESS STUDY

EMMANUELLE ANCEAUME MARIA GRADINARIU
VINCENT GRAMOLI ANTONINO VIRGILLITO

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Self-Adjusting Atomic Memory for Dynamic Systems based on Quorums On-The-Fly. Correctness Study

Emmanuelle Anceaume Maria Gradinariu
Vincent Gramoli Antonino Virgillito*

Systèmes communicants
Projet Adept

Publication interne n1795 — Avril 2006 — 15 pages

Abstract: Atomic memory is a fundamental building block for classical distributed applications. The new emergent large scale applications such as e-auctions or e-commerce need similar fundamental abstractions able to cope with the highly dynamicity of p2p environments. In this paper we prove the correctness of a Self-Adjusting Atomic Memory that implements multi-writer/multi-reader atomic operations in dynamic systems. The architecture of this system was introduced in [5, 6]. The self-healing property guarantees the capability to cope with replica volatility (unavoidable in a p2p environment), while the self-adjusting property helps it to adapt on-the-fly to the dynamicity of client access.

Key-words: Distributed algorithm, atomicity, self-adjusting, self-healing, replication, dynamism, scalability, quorums on the fly.

(Résumé : tsvp)

* Università di Roma, “La Sapienza", Italy

Correction de SAM : Une mémoire atomique auto-ajustable pour les systèmes dynamiques basée sur des quorums à la volée

Résumé : Ce papier propose la preuve de correction de SAM - une mémoire atomique auto-ajustable basée sur l'utilisation de systèmes de quorums. L'architecture de SAM a été décrite dans [5, 6]. Nous détaillons également un des modules du système qui permet de réaliser la consultations "à la volée" des quorums en lecture ou écriture.

Mots clés : atomicié, auto-ajustement, réplication, scalabilité, dynamicité, quorums à la volée

1 Introduction

The notoriety of peer-to-peer (p2p) file sharing guarantees the subsequent success of this technology in commercial applications. Some of these applications require strong computational guarantees. Internet-scale applications such as *e-auction* or *e-booking* for example, require linearizable read/write operations. That is, an auctioneer must be able to read other bids as well as writing its own bid. Alternatively, booking transaction needs persistent record. More generally, these applications need an atomic memory service able to cope with the dynamic nature of the system.

Designing an atomic memory service 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 dynamicity of the network has an important 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 replicas. The former problem can be solved using *self-healing* techniques while the latter finds solutions in the use of *dynamic quorums* (intersecting sets).

Another issue posed by the dynamicity of the network is the replica stress (load). An inadequate number of replicas may have tremendous 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.

Related Work Starting with Gifford's weighted votes [11], quorum systems [3, 16, 8, 28] have been widely used to provide consistency. Several quorum-based approaches provide mutual exclusion [22] or shared memory emulation [7]. Recently, quorum-based implementations of atomic memory for dynamic systems have been proposed in [17, 9, 12]. All these papers have a common design seed—they use reconfigurable quorum systems, work pioneered by Herlihy [14], for static distributed systems. In such systems, clients have to know the set of replicas, and thus have to participate to the reconfiguration process. In [20, 10] the authors showed that using two quorums systems concurrently preserves atomicity. This result has been later 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).

The authors in [5] follow an alternative approach for implementing atomic memory in dynamic systems. This approach is based on recent achievements in the context of dynamic quorums and logical overlays. Dynamic quorums have been mainly investigated in [26, 2, 23]. Naor and Wieder [26] sought solutions for deterministic quorums using dynamic paths in a planar overlay [24]. 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 [28], 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 replica joins and leaves and quorum sets caching in order to reduce the access latency. AndOr strategies [25] are studied in [23] in order to implement fault-tolerant storage in dynamic environment. In [21], the authors propose a solution that ensures atomic access to non replicated data in a CAN network using restrictive hypothesis: nodes nicelly leave the system by notifying their neighbours. It is also assumed that nodes do not fail and the network is reliable.

Our Contributions In this paper we propose and prove the correctness of SAM, an atomic memory for dynamic systems with self-adjusting and self-healing capabilities. SAM construction brings together several new and old research areas exploiting the best of these worlds: logical overlays, dynamic quorums and replica control. SAM architecture, which is detailed in [5], is composed of three interconnected modules, each being designed to serve a specific task (i.e., data availability, linearizability and load balancing of replicated data). Object availability is guaranteed through replication among several system nodes. Replicas of the same object define a torus overlay, structure that has been proved efficient in the design of quorum systems ([15, 25]), that is, a system made of a set of subsets of nodes, such that every two of which intersects. The behavior of the atomic memory is emulated through dynamic quorum sets sampled from a deterministic overlay traversal. Unlike quorums based on dynamic paths [26] which avoid the holes in the overlay by following alternative longer paths, our traversal fully exploits the self-healing capabilities of the overlay in order to probe a minimal number of nodes.

We propose an algorithm that implements such quorum-based phases, and prove that the read/write operations are atomic—i.e., linearizable—despite asynchronism, node arrivals, node failures, and message losses. A 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. Furthermore, our algorithm guarantees self-adjustment by expanding or shrinking the overlay with respect to the dynamicity of client accesses. We show that atomicity is not broken even when new replicas have to be added or removed from the overlay. Our work is hybrid between the reconfiguration-based systems and strategic adaptive systems. Indeed, clients or replicas do not have to be aware of the reconfiguration process. Only lightweight reconfiguration is used to achieve self-healing and self-adjusting properties, which is further exploited to design adaptive strategies for sampling read/write quorums, and balancing the replicas stress. All these properties—atomicity, self-adjustment and self-healing—are guaranteed using only constant size local information and, as far as we know, no atomic memory has been proved correct in such circumstances.

Road Map The paper is organized as follows. Section 2 introduces the system model and the problem definition. An overview of SAM is presented in Section 3. The module that handles read and write atomic operations is formally described in Section 4, and then proofs of correctness are presented in Section 5. In Section 6, we conclude and present some future research topics.

2 Model and Problem Definition

System. We consider a dynamic system $\mathcal{DS}$ as the tuple $\mathcal{DS} = (I, X)$, where I is a set of possibly infinite nodes, and X is an unbounded universe of shared data, referred in the following as *objects*. $\mathcal{DS}$ is subject to unpredictable changes. Nodes can leave or join the system arbitrarily often. Departure includes crash failure while recovery of a failed node is treated like a new arrival. A node is called *active* if it has joined and did not fail or leave the system since then. Otherwise the node is called *failed*. For each node, a *local history* is defined as a sequence of *events*, which consists of *connect*, *disconnect*, *sending* a message to other nodes, *receiving* a message from another node, taking computation steps, and possibly a *crash* event. If a crash event exists, it is always the last event in the local history. A collection of local histories, one for each node, is called an *execution*. Only *well formed executions* are considered, that is, if a message is received in the execution, then it was also sent during this execution.

Each object $x \in X$ has a single owner node and is replicated at some other nodes in the system. By abuse of notation, a replica of x refers to a node maintaining a copy of x. For efficiency reasons the number of replicas of an object is strictly inferior to the number of nodes in the system. Each object has a *value* that can be consulted/modified as a result of a read/write operation. It is assumed that each object has a predefined initial value. In order to read/write the value of an object a client node starts probing a set of this object's replicas. A read/write operation starts when the first replica in the set is contacted and completes when all the replicas in the set has been contacted.

Self-adjusting and self-healing atomic memory. This work aims at providing a self-adjusting and self-healing atomic memory. Atomicity is often defined in terms of an equivalence with a serial memory. In the following, the definition proposed in [19] is adopted:

Definition 1 (Atomicity) *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:*

- *No operation has infinitely many other operations ordered before it;*
- *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 $\pi_2 \prec \pi_1$;*
- *All write operations are totally ordered and every read operation is ordered with respect to all the writes;*
- *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.*

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 at ensuring the availability of an object whenever failures occur while self-adjustment aims at expanding or restraining the number of replicas with respect to the dynamicity of the memory load. The load of a memory is defined regarding to the load of each of its replica in the following manner:

Let $\mathcal{L}_i(t)$ be the number of operations that some replica i has to process at time t. $\mathcal{L}_i(t)$ is referred in the following as the local load of replica i at time t. Let T^i_{min} and T^i_{max} be two application dependent parameters which define a lower and upper bound on the load of i. Replica i is *overloaded* (resp. *underloaded*) iff $\mathcal{L}_i(t) \geq T^i_{max}$ (resp. $\mathcal{L}_i(t) \leq T^i_{min}$).

Definition 2 (Self-adjusting Atomic memory) *Let $DS = (I, X)$ be a dynamic system and α be an execution of DS. Let x be an object in X and let M_x be the set of x replicas. For each i in M_x let T^i_{min} and T^i_{max} the application dependent load boundaries. M_x is a self-adjusting atomic memory with respect to the load iff:*

- **atomicity** *α restricted to M_x verifies the atomicity definition (see Definition 1)*
- **self-adjustment** *$\forall i \in M_x, \exists t_{0_i}$ such that $\forall t > t_{0_i}, T^i_{min} \leq \mathcal{L}_i(t) \leq T^i_{max}$.*

3 SAM Overview

SAM aims at emulating an atomic memory on top of a replicated system with self-adjusting and self-healing capabilities. This section presents an overview of SAM.

3.1 SAM Logical Overlay

Replicas of an object, referred in the following as memory, share a same logical overlay, organized in a torus topology (as for example CAN [27]). Basically, a 2-dimensional coordinate space $[0, 1) \times [0, 1)$ is shared by all the replicas of an object. A replica is responsible of a set of zones, each of those being a rectangle in the plane. The entrance and departure of a replica dynamically changes the decomposition of the zones. These zones are rectangles (union of rectangles) in the plane. Replicas owners of adjacent zones 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 [27, 26]: 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. The object initial value v_0 is replicated at 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 SAM Architecture

SAM is specified in Input/Output Automata (IOA) Language [18] and is structured as the composition of three main automata: the $Traversal_{x,i}$, the $LoadBalancer_{x,i}$, and the $Adjuster_{x,i}$ automata, where $i \in I$ and x is the considered object. A fourth automaton, the $CommunicationLink_{x,i,j}$, represents the communication medium between any two replicas i and j. In the remaining of the paper, we restrict our attention to only one object x. Thus, the x subscript is omitted. Relationship among the four automata in terms of input/output actions is depicted in Figure 1.

Briefly, the LoadBalancer automata scans the overlay to identify non overloaded replicas. It determines the strategies for reducing the memory load at nodes, and includes an operation aggregation strategy to overlap operation executions. The Traversal automaton is in charge of maintaining the consistency of the overlay applying appropriate strategies for executing linearizable operations on the replicas. Finally, the Adjuster assigns logical responsibility zones to physical replicas and maintains this correspondence consistent. It also handles the expansion or shrink of the overlay whenever requested by the Load Balancer, and the departure of non responding replicas.

Replicas are accessed by clients through read and write operations on the object. Such operations are implemented in the Traversal automaton. A read operation consists in traversing the overlay following a horizontal trajectory that wraps around all the overlay. A write operation consists in traversing the overlay following a horizontal trajectory and then a vertical one. An horizontal traversals defines a *consultation quorum*, while a vertical one is a *propagation quorum* (see Figure 2). More details are given in the following Section.

SAM starts with a read/write request from a client. A client submits a read/write request to one of the replicas. The Load Balancer of this replica does not immediately initiate a traversal but rather records this request in a local

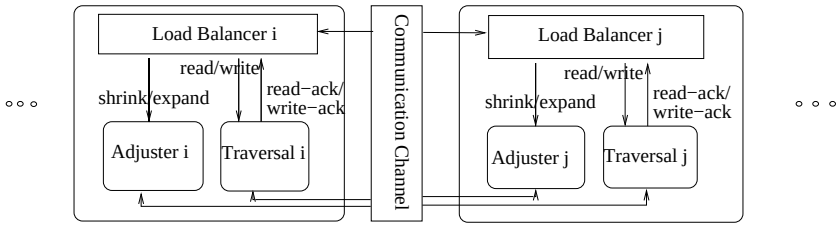

Figure 1: Overview of SAM

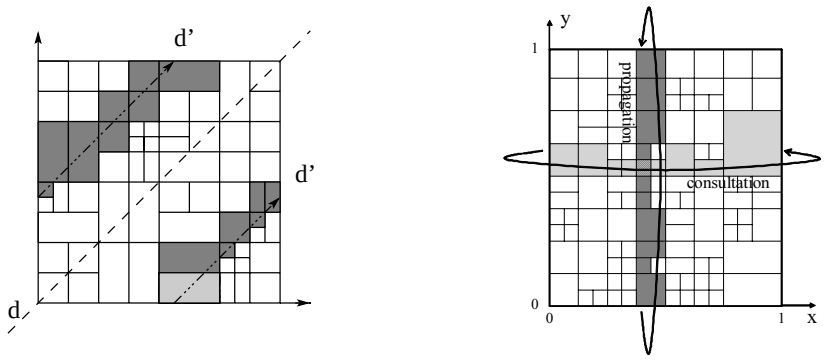

Figure 2: A Thwart Path (left grid) and a Traversal Path (right grid)

heap. Each replica periodically scans its heap to pick the requests for which a traversal has to be initiated. The strategy to pick these requests consists in choosing only one write (or read) operation among all the pending ones. More precisely, a write operation is initiated only for the most recently heaped write request (if any), while a read traversal is initiated for one of the heaped 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 a traversal is initiated, the initiating replica empties its heap, after having kept track of all the read/write requests to later return the status of the operations to the requesting clients. Clearly, this request aggregation strategy reduces the number of requests that are actually served and thus the latency of the memory.

Despite the former aggregation strategy, the number of heaped requests at an initiating replica still can grow very fast, incurring in a local overload as defined in Section 2. If a replica load catches up with a threshold T_{max} then the Load Balancer invokes a *thwart* process to find a suitable replica. The thwart process, as shown in Figure 2 shown in the appendix, checks the memory along a diagonal trajectory for a non overloaded replica. The diagonal trajectory is chosen to contact at each step a replica involved in different traversals. If such a replica is found, then it becomes the initiating replica for these requests, otherwise if the memory needs to be expanded (no overloaded replica has been found), the Load Balancer activates the Adjuster to add another replica (i.e., , it requests a node in the system to host a replica, hence to become a SAM member [1]).

Alternatively, when the load of a replica is below T_{min}, the Load Balancer invokes the Adjuster for a shrink procedure: the replica is removed from the memory. This prevents probing complexity from remaining uselessly high when a burst of requests has ended. When a replica leaves (voluntarily 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). A complete specification of the Load Balancer and Adjuster is given in the Appendix.

[1] As mentioned in the previous section, the current work does not focus on mechanisms for motivating nodes to host replicas

4 The Traversal Automaton

The Traversal is the core of SAM activity. It is in charge of maintaining the consistency of the replicas by applying appropriate strategies for executing linearizable operations via intersected sets, called *quorums*. In this Section we give a complete specification of the Traversal automaton.

4.1 Read/Write Operations

SAM guarantees the atomicity of read/write operations using a quorum system referred in the following as the *tiling quorum system*. The key notions in the definition of a tiling quorum system are the *horizontal* and *vertical* tiling sets. A set of replicas in SAM define an horizontal (resp. vertical) tiling if their responsibility zones are pairwise independent and totally cover the abscissa (resp. ordinate) of the coordinate space shared by replicas in SAM.

Definition 3 (Tiling Quorum System) *A tiling quorum system $\mathcal{T}$ is a couple $\langle \mathcal{C}, \mathcal{P} \rangle$ where $\mathcal{C}$ is a set of horizontal tilings and $\mathcal{P}$ a set of vertical tilings such that any horizontal tiling intersects any vertical tiling.*

Let $\mathcal{T} = \langle \mathcal{C}, \mathcal{P} \rangle$ be a tiling quorum system. Elements in $\mathcal{C}$ are referred to in the following as *consultation quorums*, while elements in $\mathcal{P}$ are referred to as *propagation quorums*. Figure 2 in the appendix shows a consultation and a propagation quorum as well as the corresponding traversals.

Read and write operations in SAM are multi-phase operations as in [7, 17, 12, 9]. While in [7, 17] a read operation is realized in two phases, SAM, similarly to GeoQuorums [9] exploits single-phased read operations, for efficiency reasons. Then, the read operation consists in only one consultation phase, where a consultation quorum is probed. That is, the overlay is traversed horizontally (from west to east) and all the nodes encountered are requested for their object *value* and *tag* (the value's timestamp). Finally, the most updated value is returned by the read. The write strategy contains the same consultation phase to get the most recent tag, plus a propagation phase, where the value to write is written on a propagation quorum with an updated version number. To differentiate the consultation phases of read and write operations, the latter is called the *update phase*.

One-phase reads might violate linearizability in the following way: a read operation executed concurrently with a write operation may consult a fresh value not completely propagated, while a following read may consult an old value. The problem comes from the early termination of a read operation that consults a fresh value which is still being propagated. This violates the atomicity definition (see Definition 1).

GeoQuorums solves this problem by including an additional phase in the write operation, in which the initiator of a write sends a specific confirmation message when the write operation is completed. Differently, in SAM propagation terminates before termination of the read. This is achieved by using a lock mechanism to let either another read operation consult this newly written value or to order this read operation before the write. More precisely, in the propagation phase the overlay is traversed in a orthogonal direction with respect to consultation (north and south). Propagation proceeds in both senses so that each replica is visited twice in this phase: the first time the object is locked, preventing concurrent reads to return the updated value too early, 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 [13]. In order to guarantee the termination of the consultation phase when it crosses an infinite sequence of propagations we use the following mechanism: each consultation arriving at a replica takes a snapshot of the locally pending propagations and waits only for those propagations to end. Hence, in a finite number of steps each consultation phase terminates.

4.2 Automaton States

The state variables of the Traversal automaton are described for node i from line 7 to 37 of I/O Automaton 1. First of all, each replica maintains a tag and a value of the object (tag and val fields). Those fields are updated when the replica participates in a write operation involving the quorums it belongs to. The op record (cf. lines 16–29) describes an operation. It contains some identifying subfields such as its identifier id, its $type$ which indicates if the operation associated is a read or a write, and $intr$ and $intr\text{-}zone$ informing about the identifier of the replica that started the operation and the starting zone. The $senses$ and $phase$ fields indicate respectively the sense (E, N or S) and the phase of the traversal in progress, while $zone$ is the zone involved in the operation, among the ones managed by the replica. Finally, two last fields are required by the lock mechanism described above: $locking\text{-}prop$ and $rcv\text{-}prop$. The

I/O Automaton 1 $OperationManager_i$

Domain:
- $I \subseteq \mathbb{N}$, the set of node identifiers.
- V, the set of all possible values of an object.
- $\Pi \subseteq I \times \mathbb{N}$, the set of all operation identifiers.
- $T \subseteq I \times \mathbb{N}$, the set of all tags.
- M, the set of all possible messages.

Signature:

Input:
- $\mathsf{read\text{-}write}(type, id, v)_i,\ i \in I, type \in \{\mathsf{read}, \mathsf{write}\}, id \in \Pi, v \in V$
- $\mathsf{rcv}(m)_{j,i},\ i, j \in I, m \in M$

Output:
- $\mathsf{snd}(m)_{i,j},\ i, j \in I, m \in M$
- $\mathsf{read\text{-}write\text{-}ack}(id, v)_i,\ i \in I, id \in \Pi, v \in V$

State:
- op a record with fields
  - $id \in \Pi$
  - $intr \in I$
  - $zone \in \mathbb{R}^4$
  - $intr\text{-}zone \in \mathbb{R}^4$
  - $type \in \{\mathsf{read}, \mathsf{write}\}$
  - $phase \in \{\mathsf{cons}, \mathsf{update}, \mathsf{prop}, \mathsf{ending}\}$
  - $rcv\text{-}prop \subseteq \{N, S, E\}$
  - $locking\text{-}prop \subseteq \Pi$, initially $\perp$
  - tag, a record with fields
    - $ct \in \mathbb{N}$
    - $id \in I$
  - $val \in V$, initially $\perp$
  - $senses \subseteq \{N, S, E\}$
- ops, the set of all known operations
- tag, a record with fields
  - $ct \in \mathbb{N}$
  - $id \in I$
- $val \in V$, initially v_0
- $failed$ a boolean, initially false
- $replica$ a boolean
- $adjusting$ a boolean

former indicates, for a consultation phase, the set of pending propagations that blocks it. The latter indicates, for a

Transitions:

Input $\mathsf{read\text{-}write}(type, id, v)_i$
- **Effect:**
  - **if** $\neg failed \wedge replica$ **then**
    - $op.id \leftarrow id$
    - $op.type \leftarrow type$
    - **if** $op.type = \mathsf{read}$ **then**
      - $op.phase \leftarrow \mathsf{cons}$
      - **for** $op1$ st. $op1.phase = prop$ **do**
        - $op.locking\text{-}prop \cup \leftarrow \{op1.id\}$
    - **else**
      - $op.phase \leftarrow \mathsf{update}$
      - $op.val \leftarrow v$
    - $op.intr \leftarrow i$
    - $op.zone \in i.zones$
    - $op.intr\text{-}zone \leftarrow op.zone$
    - $op.senses \leftarrow \{E\}$
    - $ops \leftarrow ops \cup \{op\}$

Input fail_i
- **Effect:**
  - $failed \leftarrow \mathsf{true}$

Output $\mathsf{snd}(\langle id, phase, intr\text{-}zone, z, t, v, sense\rangle)_{i,j}$
- **Precondition:**
  - $\neg failed \wedge replica \wedge freezing = \emptyset$
  - $op.locking\text{-}prop = \emptyset$
  - $op \in ops$
  - $phase = op.phase \notin \{\mathsf{ending}, \mathsf{idle}\}$
  - $id = op.id$
  - $\langle t, v\rangle = op.\langle tag, val\rangle$
  - $sense \in op.senses \neq \emptyset$
  - $intr\text{-}zone \leftarrow op.intr\text{-}zone$
  - $z = \mathsf{get\text{-}trv\text{-}zone}(sense, intr\text{-}zone, op.zone)$
  - $j = \mathsf{get\text{-}responsible}(z)$
- **Effect:**
  - $op.senses \leftarrow op.senses \setminus \{sense\}$

Input $\mathsf{rcv}(\langle id, phase, intr\text{-}zone, z, t, v, sense\rangle)_{j,i}$
- **Effect:**
  - **if** $\neg failed \wedge replica \wedge phase \neq \mathsf{idle}$ **then**
    - $op = op' \in ops$ st. $op'.id = id$ or $\perp$
    - $op.phase \leftarrow phase$
    - $op.zone \leftarrow z$
    - **if** $op.phase = \mathsf{cons}$ **then**
      - $op.\langle tag, val\rangle \leftarrow \mathsf{max}(\langle tag, val\rangle, \langle t, v\rangle)$
      - **for** $op1 \in ops : op1.phase = \mathsf{prop} \wedge |op1.rcv\text{-}prop| = 1$ **do**
        - $op.locking\text{-}prop \cup \leftarrow \{op1.id\}$
      - **if** $op.intr = i$ **then**
        - $op.phase \leftarrow \mathsf{ending}$
    - **else if** $op.phase = \mathsf{update}$ **then**
      - $op.tag \leftarrow \mathsf{max}(tag, t)$
      - **if** $op.intr = i$ **then**
        - $op.phase \leftarrow \mathsf{prop}$
        - $op.tag \leftarrow \langle op.tag.ct + 1, i\rangle$
        - $op.senses \leftarrow \{N, S\}$
    - **else if** $op.phase = \mathsf{prop}$ **then**
      - $\langle tag, val\rangle \leftarrow \langle op.tag, op.val\rangle$
      - $op.rcv\text{-}prop \leftarrow op.rcv\text{-}prop \cup \{sense\}$
      - **if** $|op.rcv\text{-}prop| = 2$ **then**
        - **for** $op2 : op2.phase = \mathsf{cons}$ **do**
          - $op2.locking\text{-}prop \setminus \leftarrow \{op.id\}$
        - **if** $op.intr \neq i$ **then**
          - $op.senses \leftarrow op.senses \cup \{sense\}$
      - $ops \leftarrow ops \cup \{op\}$

Output $\mathsf{read\text{-}write\text{-}ack}(id, v)_i$
- **Precondition:**
  - $op \in ops$
  - $op.phase = \mathsf{ending}$
  - $id = op.id$
  - $v = op.val$
  - $op.intr = i$
- **Effect:**
  - $op.phase \leftarrow \mathsf{idle}$

propagation phase, the senses from where messages have been received. In overall, the number of elements in this set informs whether or not the propagation phase is pending.

Messages in a traversal are sent from neighbor to neighbor, starting at the initiator replica $op.intr$ of the operation. The messages sent during the traversal contain the operation identifier id, type $type$, initiator zone $intr\text{-}zone$, but also information about where it has to be sent (regarding to $sense$ and current zone z). Other fields that are related to the Traversal automaton are the booleans $failed$ and $replica$, indicating respectively whether the current node is crashed or not and whether it owns a copy of the object or not, and the set ops containing the operations the replica keeps track of. Finally, the $adjusting$ field is needed in case of memory adjustment, due to nodes joining or leaving the memory. If it is true, this indicates that an adjustment is pending at the current node, and no traversal participation for that node is possible before it recovers its state.

4.3 Automaton Transitions

The transitions of the $Traversal$ automaton are described for node i in I/O Automaton 1, lines 38–109. A read or write operation starts at node i in the consultation phase as a result of a $\mathsf{read\text{-}write}_i$ input event. This event is triggered by the LoadBalancer automaton and sets the op subfields. More precisely it initiates the starting phase (update or cons) sense to E, the initiator identity, the current zone ($zone$), the initiator zone ($intr\text{-}zone$), and the $locking\text{-}prop$ field if any propagation phase are pending at the current replica. Notice that it is the responsibility of the $LoadBalancer$ to assign a new unique identifier to this operation, whereas the former event initializes other operation subfields.

The traversal proceeds with messages being sent by a node i to one of its east neighbors j, through a $\mathsf{snd}_{i,j}$ event. Such an event is triggered only if i is not $adjusting$, $failed$ and is a $replica$. Note that if the $op.locking\text{-}prop$ is not empty, then the snd can never occur (cf. precondition line 62). This means that even if a replica has received a message corresponding to a traversal, while it is failed, blocked or adjusting, it does not participate by sending new messages.

When a replica i receives a message from a $Traversal_j$ automaton by a $\mathsf{recv}_{j,i}$ input event, it checks (i) if it is the initiator of the operation ($op.intr = i$) and (ii) the current phase of this operation. Let ϕ be this phase.

When all the phases of an operation are done, the initiator sets the $op.phase$ subfield to ending. This occurs when the initiator of the operation receives the consultation message from the east direction, or when it has received propagation messages from both north and south senses ($op.rcv\text{-}prop = \{N, S\}$). For an update phase to complete, the initiator receives an east message and changes $op.phase$ to prop. There is a singular difference between propagation phase and update/consultation phases in the way tag and value are altered.

In all cases the replica receiving a message, updates either the operation $op.\langle tag, val\rangle$ pair or it updates its local $\langle tag, val\rangle$ pair. The most up-to-date pair is conveyed by messages as the $\langle t, v\rangle = op.\langle tag, val\rangle$ pair (l. 66). In case ϕ is a consultation or an update phase, the tag information is simply the most up-to-date one encountered: it is the maximum between the local one and the received one (l.80, 87). If ϕ is a consultation phase, the chosen value is the value related to this tag, whereas if ϕ is an update phase, the value remains the one initially set as the value to write. Finally if ϕ is a propagation then the local pair is updated with the received one (l. 93), since the value included is the value to be written.

Next, a consultation receipt at a replica i might make this consultation block. More precisely if there are pending propagations at i when the consultation receipt occurs (l.81–83), the operation $locking\text{-}prop$ field is filled which prevents further snd event from occurring. From this point on, each time one of the pending propagations completes, an element is removed from the $locking\text{-}prop$ set (l.97). When this set becomes empty again, the snd event is newly enabled, provided that i is still an active non-adjusting replica.

Moreover, the $op.sense$ subfield is set to the sense of the phase message and op field (including all its subfields) is stored in ops (l.100). Each replica sends messages with information about operations op in a sense belonging to $op.senses$ set by the mean of the snd action. Since an initiator starting a consultation or an update phase has its op field set to $\{E\}$, the first message is sent in the east sense. When a non-initiator participating replica receives the phase message, it records the op subfields to resend it simply in the same sense. If a propagation phase starts at an initiator, this one must send two messages: one in the north sense and the other in the south sense.

Note the presence of the failed input action. This action is triggered by the environment and indicates that a crash has occurred. Once this flag is set to true any other action is disabled.

Finally, we briefly describe two functions that are not specified in the $Traversal$ automaton. Function get-trv-zone which, given a sense (N,S,E), the traversal initiator zone, and the operation current zone $op.zone$, returns the next zone in this sense belonging to one of the possible trajectory of the traversal (this zone is crossed by an horizontal line

also crossing the initiator zone). If many possible replicas exist it chooses one among all. Function get-responsible which, given a zone, returns the neighbor responsible for this zone.

5 SAM Correctness

This section proves SAM's correctness. Hence, we show that SAM is a self-adjusting atomic memory as defined in Section 2. This proof is divided in three parts. First, Theorem 5.1 shows that operations contact tiling quorum of replicas. Second, the atomicity is proved in Theorem 5.6. Finally, we prove that SAM self-adjusts (Theorem 5.7).

Assumptions. The $never\text{-}partitioning$ property ensures that a set of connected nodes is never partitioned, while the $failure\text{-}detection$ property ensures that a failure is detected without any mistake. We assume that those two conditions are satisfied by our overlay of replicas.

We do not show explicitly how a replica gets "a failed replica locking-prop state back". When a replica fails and a new replica takes over its zone, or when a new replica is created the corresponding active node needs to get back *locking-prop* information. Given the $never\text{-}partitioning$ property, the corresponding replica contacts its north and south neighbors to get the highest tag, val, $locking-prop$, $senses$ values, and information about their phase participation before participating in any message exchange. The choice of an healing replica is done using CAN overlay mechanism and is not formally specified here.

We also 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 a $\mathsf{read\text{-}write}_i$ event or a fail_i event; (ii) a $\mathsf{read\text{-}write}_i$ event is immediately followed by the matching $\mathsf{read\text{-}write\text{-}ack}_i$; (iii) no fail_i event precedes any $\mathsf{read\text{-}write}_i$, or $\mathsf{read\text{-}write\text{-}ack}_i$ event.

For the sake of stabilization of our approach, we assume the following two properties: (i) the $potential\text{-}replication$ property ensures that there exists at least one active node out of the memory at any time; (ii) the $thwarting\text{-}access\text{-}rate$ property means that during any thwart, the access rate (rate at which requests are received) is lower than the logical treatment rate (rate at which requests are started). If it exists a replica accepting to treat the thwarted request then the duration time of the thwart is the period between the first sent in the thwart and the acceptation time. Otherwise, it is the period between the first sent and the time the expand resulting from the last receipt (when the thwart has wrapped around the torus) occurs.

Definitions. We refer to **tag** as a mapping from TId to $\mathbb{N} \times I$ such that $op.id$ is mapped to $tag(op.id)$ if at least one of the following condition holds: (i) if $op.type = \mathsf{read}$ then $tag(op.id) = tag_i$ when $\mathsf{read\text{-}write\text{-}ack}_i$ occurs, (ii) if $op.type = \mathsf{write}$ then $tag(op.id) = tag_i$ immediately after $\mathsf{rcv}(\langle *, \mathsf{prop}, ...\rangle)_{j,i}$ occurs. Given this definition, we totally order operations by their tag, respecting the lexicographical order.

We define **inv** as a mapping from an operation to $\mathbb{R}^+$ such that operation π is mapped to time τ if $\mathsf{read\text{-}write}(*, id, *)$ occurs at time τ with $\pi.id = id$.

We define **resp** as a mapping from an operation to $\mathbb{R}^+$ such that operation π is mapped to time τ' if $\mathsf{read\text{-}write\text{-}ack}(id, *)$ event occurs at time τ' where $\pi.id = id$.

Next, 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 $resp(\pi_1) \prec inv(\pi_2)$.

Finally, we restate the **atomicity** definition of Lynch [19] using the aforementioned notations. If it exists a matching $\mathsf{read\text{-}write\text{-}ack}_{x,i}$ event following any $\mathsf{read\text{-}write}_{x,i}$ event then, it exists a relation $<_S$ that orders partially read and write operations such that: (i) $<_H \subseteq <_S$. (ii) For any write operation π_1 and π_2, either $\pi_1 <_S \pi_2$ or $\pi_2 <_S \pi_1$. (iii) For any read operation π_1 and write operation π_2, either $\pi_1 <_S \pi_2$ or $\pi_2 <_S \pi_1$. Moreover, if it exists π_2 such that $\pi_2 = max_{<_S}\{\pi_2' <_S \pi_1\}$ and π_2 writes value v then π_1 returns the same value v. And if no such π_2 exists, then the value returned by π_1 is v_0. (iv) For any read or write operation π, the set $\{\pi' : \pi' <_S \pi\}$ is finite.

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 $tag(\pi_1) < tag(\pi_{\ell'})$.

Quorums properties. In this section we prove that for a given phase, our traversal mechanism involves a quorum of nodes. That is the phase set of participating replicas verifies each of the properties defining a quorum. First of all, we say operation π's phase completes in one of the three following cases holds: (i) $\mathsf{rcv}(\pi.id, \mathsf{cons}, ...)$ occurs and $op.phase$ is set to ending, (ii) $\mathsf{rcv}(\pi.id, \mathsf{update}, ...)$ occurs and $op.phase$ is set to prop, or (iii) $\mathsf{rcv}(\pi.id, \mathsf{prop}, ...)$ occurs and $op.phase$ is set to ending. We define the initiator of an operation π the node where π starts. Typically i is the initiator of operation π if $\mathsf{read\text{-}write}(*, \pi.id, *)$ occurs at node i.

Theorem 5.1 *If a phase ϕ completes, all elements of a quorum Q have received messages such that:*
- *if ϕ is a propagation phase then Q is an propagation quorum.*
- *if ϕ is a consultation phase or an update then Q is a consultation quorum.*

Proof. First we show that all messages of ϕ are sent in a non-varying sense. When ϕ starts, the initiator sends messages in each sense included in the $senses$ set. A non-initiator replica changes its field only by a rcv (or by a snd event to make sure messages are not resent) and it adds the receipt sense to its $senses$ set. The initiator $senses$ is set or updated only when a new phase starts: $phase$ is set to cons or update and $senses$ to $\{E\}$, or $phase$ is set to prop while $senses$ is set to $\{N, S\}$, or $phase$ is set to ending and prop or ending.

Second, we show that replicas are contacted following a chain of adjacent zones. By examination of the snd action, if a $\mathsf{snd}(*, *, *, z, *, *, sense)_{i,j}$ event occurs then z belongs to $j.zone$. Next, by $\mathsf{get\text{-}trv\text{-}zone}_i$ function, we know that zone z is chosen such that it abuts the current zone $op.zone_i$ along the $sense_i$ axis. Lastly, the corresponding $\mathsf{rcv}(*, *, *, z, *, *, sense)_{i,j}$ sets the current phase to z received since j is an active replica.

From now on, we show that it exists a common ordinate (resp. abscissa) to every zone, the consultation or update (resp. propagation) phase goes through. The proof follows directly from the $intr\text{-}zone$ field and the definition of the get-trv-zone function. First, notice that field $op.intr$ is initially set to the current node zone by a read-write event and is not modified in any further state. By definition of get-trv-zone and the snd action, the next participant's zone is such that it exists a horizontal (resp. vertical) line that crosses it and that also crosses the $op.intr\text{-}zone$.

All these three properties leads to the conclusion. If ϕ is a consultation, the sense followed by messages is E, all zones crossed over are adjacent, and there exists a common abscissa between them, i.e., , ϕ contacts a set of participants consisting of a horizontal tilling. The vertical one is shown following the same reasoning. The intersection is straightforward from tiling definition. □

Atomicity proof. Here, we show that the partial order relation $<_t$ exists. The first lemma shows that no locked replica can participate in any read operation until it is unlocked. The second one shows that a replica receives the second propagated message after all participants have already received the first one.

Lemma 5.2 *If $|\pi_1.rcv\text{-}prop_i| = 1$ at time τ such that $\mathsf{rcv}(\pi_2.id, \mathsf{cons}, *, *, *, *, *)_i$ occurs at time $\tau' > \tau$ with $i \neq \pi_2.intr$, every participant or π_1's propagation have already stored the propagated value, tag pair at time $resp(\pi_2)$.*

Proof. When $\mathsf{rcv}(\pi_2.id, \mathsf{cons}, *, *, *, *, *)_i$ occurs, replica i knows about operation π_1. By assumption, it exists $\pi_1 \in ops$ such that $|\pi_1.rcv\text{-}prop_i| = 1$. By examination of the code, since $\pi_2.phase_i = \mathsf{cons}$, the π_1 identifier is added to $locking\text{-}prop_i$ subfield of operation π_2. Observe that no other action modifies the subfield $\pi_2.locking\text{-}prop_i$. Next, notice that the initiator of a consultation phase needs to get back the message it initially sent, before the phase can terminate. By assumption $i \neq \pi_2.intr$. Thus i must send a message before $\pi_2.intr$ receives it. No snd_i event can, however, occur until $\pi_2.locking\text{-}prop_i$ is not empty. The $\pi_2.locking\text{-}prop_i$ subfield is emptied only if i receives a second message corresponding of every pending propagations whose identifiers belong to $\pi_2.locking\text{-}prop_i$. 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 of π, it contacts its two neighbors in its propagation quorum. 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. Therefore, each of those have already set their value and tag pair to the one received when $\pi_2.locking\text{-}prop_i$ is emptied. Since the snd_i is enabled for consultation π_2 after this time, $resp(\pi_2)$ occurs when all participants of π_1 have stored the propagated tag and val pair. □

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

Lemma 5.3 *If* $resp(\pi_1) \prec inv(\pi_2)$ *then* $tag(\pi_1) \leq tag(\pi_2)$.

Proof. In the first case assume that π_1 is a write operation, hence π_1 completes only if its propagation phase has already ended. If $resp(\pi_1) \prec inv(\pi_2)$, it means that π_1 has propagated its tag to a whole propagation quorum, when operation π_2 is initiated. By the quorum intersection property, it exists one element j that gets assigned this tag in every consulting quorum by time $resp(\pi_1)$. Since tag value is monotonically incremented, and the assumptions ensure that any failure is followed by a state recovery, it is clear that tag tag', consulted during the consultation phase π_2, is such that $tag' \geq tag(\pi_1)$. By definition of $tag(\pi_2)$, it is such that $tag(\pi_2) \geq tag'$ and putting these inequalities together yields to the result.

Now consider the second case where π_1 is considered to be a read operation. To prove that the property holds, we show that the following contraposition is true: If $tag(\pi_2) < tag(\pi_1)$ then $start(\pi_2) \prec term(\pi_1)$. Assume that π_1 contacts the consulting quorum c_1 while π_2 contacts the consulting quorum c_2. Hence if $tag(\pi_2) < tag(\pi_1)$ then $\exists j \in c_1$ such that $j.tag > max_{i \in c_2}\{i.tag\}$. Given that, we show that $resp(\pi_1) \prec inv(\pi_2)$ is impossible. The existence of j implies that it exists a write operation π_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 π_3 will eventually propagate to one element of c_2. By lemma 5.2, j is locked until after having propagated to an element of c_2 and $resp(\pi_1) \prec inv(\pi_2)$ is impossible: if $tag(\pi_2) < tag(\pi_1)$ then $inv(\pi_2) \prec resp(\pi_1)$ and the result is also true, i.e., if $resp(\pi_1) \prec inv(\pi_2)$ then $tag(\pi_1) \leq tag(\pi_2)$. □

Corollary 5.4 *If* $resp(\pi_1) \prec inv(\pi_2)$ *and* π_2 *is a write traversal then* $tag(\pi_1) < tag(\pi_2)$.

Proof. This follows directly from Lemma 5.3 and the definition of write operation tag. Let tag_c be the tag at the end of consultation phase of π_2 before being incremented. By examination of $\mathsf{rcv}(\pi_2.id, phase, *, *, *, *, *)$ action that ends the consultation or update phase, we conclude that $tag(\pi_2) > tag_c$. Combining this with Lemma 5.3 leads to the conclusion. □

The main theorem shows that the traversal ordering satisfies the atomicity definition.

Theorem 5.5 *The* $Traversal$ *automaton implements atomic object.*

Proof. Two write operations get assigned different tags. This follows from the fact that two writes at the same location get assigned a different sequence number, 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 $\pi_1 <_H \pi_2$ and $\neg(\pi_1 <_t \pi_2)$. Now there are two cases: (i) If π_2 is a read operation, then $\pi_1 <_H \pi_2$ and Lemma 5.3 implies that $tag(\pi_1) \leq tag(\pi_2)$. (ii) If π_2 is a write operation, then $\pi_1 <_H \pi_2$ and Corollary 5.4 implies that $tag(\pi_1) < tag(\pi_2)$. By definition of $<_t$, both results yield a contradiction. For Part 4, since any operation π_1 in H terminates, Lemma 5.3 implies that all operations π_2 such that $\pi_1 <_H \pi_2$ is ordered after. That is, the set of operations preceding π_1 is finite. Part 3 is straightforward. □

Corollary 5.6 SAM *implements atomic object.*

Proof. Let $inv_{LB}(\pi)$ be the time at which the $LoadBalancer$ automaton starts operation π, and let $term_{LB}(\pi)$ be the time at which it ends the same operation. On the first hand, $inv_{LB}(\pi) \prec inv(\pi)$, since the traversal start is ordered by the $LoadBalancer$. On the other hand, $term(\pi) \prec term_{LB}(\pi)$ since the traversal terminates before the termination operation is acknowledged. Therefore we can map any operation of the $LoadBalancer$ to the corresponding one of the $Traversal$ while choosing the same serialization point for each of them (i.e., keeping the same order among operations). By assumption, tag and value are simply updated by the Adjuster, thus non-violating atomicity. □

Self-Adjustment. We show that SAM guarantees self-adjusting property.

Theorem 5.7 *SAM self-adjusts.*

Proof. Let τ_{thw} be the duration time of thwart thw. When a replica i is overloaded, ($\mathcal{L}_i(t) > T^i_{max}(t)$) the $LoadBalancer_i$ orders the start of thwart process thw. By $thwarting\text{-}access\text{-}rate$ assumption, the access rate of i remains lower than its request treatment, thus $\mathcal{L}_i(t)$ does not increase. When the thwart terminates there are two cases: (i) if a replica j decides to treat the requests then the amount of load sent through the thwart is distributed by i to j. Since this amount is decided by i to be the set of requests over its threshold $T^i_{max}(t)$, i is no more overloaded. (ii) if no replica accept to treat the requests, then i receives back the thwart message while its $\mathcal{L}_i(t)$ has not increased since the beginning of the request. Then *potential-replication* ensures that an expand is possible and the load overhead of i is given to the new replica. Finally assume that replica i is underloaded. Then it decides to make a shrink. By leaving the memory, i is no more considered as a replica and its low load is no more taken into account. □

6 Conclusions and Future Work

In this paper we have proved the correctness of SAM, an architecture emulating an atomic memory in highly dynamic systems exhibiting self-healing and self-adjusting properties depending on object load variations and dynamicity of the system. Properties of SAM (atomicity, self-healing and self-adjusting) are implemented using only local information whose size remains low while the system size grows. SAM provides an efficient solution to the persistent storage problem defined in [4]. It is a fundamental abstraction suitable as a building block for many distributed complex applications, especially in dynamic systems.

We are currently working on experimental results of SAM based on p2p simulator to emphasize its behavior in face of dynamism and high-scale. Future work includes latency comparison of other alternative primitives for read operations. Finally, we intend to borrow some interesting results from the mechanism theory to find incentive strategy for data replication among peers.

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] E. Anceaume, M. Gradinariu, V. Gramoli, and A. Virgillito. P2p architecture for self∗ atomic memory. In *Proc. of the 8th Int. Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2005. To appear.

[6] E. Anceaume, M. Gradinariu, V. Gramoli, and A. Virgillito. Sam: Self* atomic memory for p2p systems. Technical Report 1717, IRISA, 2005.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

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

[20] 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.

[21] Nancy A. Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in distributed hash tables. In *IPTPS*, pages 295–305, 2002.

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

Appendix

A The Load Balancer and Adjuster Specification

I/O Automaton 2 $LoadBalancer_i$

Signature:

Input:

$\mathsf{rcv}(\langle \mathsf{op}, type, v\rangle)_{j,i},\ i \in I, type \in \{\mathsf{read}, \mathsf{write}\}, v \in V$

$\mathsf{read\text{-}write\text{-}ack}(id, v)_i,\ i \in I, v \in V$

$\mathsf{rcv}(m)_{j,i},\ i, j \in I, m \in M$

Internal:

$\mathsf{thwart}_i,\ i \in I$

Output:

$\mathsf{snd}(\langle \mathsf{op}, type, v\rangle)_{i,j},\ i \in I, type \in \{\mathsf{read}, \mathsf{write}\}, v \in V$

$\mathsf{read\text{-}write}(type, id, v)_i,\ i \in I, type \in \{\mathsf{read}, \mathsf{write}\},$
$id \in \Pi, v \in V$

$\mathsf{shrink}_i,\ i \in I$

$\mathsf{expand}(ra, wa)_i,\ i \in I, ra, wa \in \mathbb{N}$

$\mathsf{snd}(m)_{i,j},\ i, j \in I, m \in M$

State:

$rqst$ a record with fields

$tid \in TId$

$type \in \{\mathsf{read}, \mathsf{write}\}$

$rqstr \in I$

$intr \in I$

$val \in V$

$tag \in \mathbb{N} \times I \times \mathbb{N}$

$phase \in \{\mathsf{idle}, \mathsf{acknowledging}\}$

$batch$, an ordered set of requests $rqst$

$val \in V$

$failed$ a boolean

$replica$ a boolean

$heap$ a set of requests $rqst$

$treating$ a set of requests $rqst$

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

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

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

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

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

$access\text{-}rate \in \mathbb{R}^{>0}$ the rate of local request reception.

$treatment\text{-}rate \in \mathbb{R}^{>0}$ the rate of local request treatment.

$fwd \in TId$

$starter \in I$

$order\text{-}expand$ a boolean

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

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

Transitions:

Input $\mathsf{rcv}(\langle \mathsf{op}, type, v\rangle)_{j,i}$
  Effect:
    if $\neg failed \wedge replica$ **then**
      if $type = \mathsf{read}$ **then**
        $r\text{-}access \leftarrow r\text{-}access + 1$
      else
        $w\text{-}access \leftarrow w\text{-}access + 1$
      $ct \leftarrow w\text{-}access + r\text{-}access$
      $rqst \leftarrow \langle\langle ct, i\rangle, type, j, i, v, \perp, \mathsf{idle}, \emptyset\rangle$
      $heap \leftarrow heap \cup \{rqst\}$
      if $access\text{-}rate \geq treatment\text{-}rate$ **then**
        $fwd \leftarrow fwd \cup \mathsf{top}(heap)$
        $heap \leftarrow \mathsf{bottom}(heap)$
        $starter \leftarrow i$
      $shrink\text{-}time \leftarrow clock + shrink\text{-}period$

Output $\mathsf{read\text{-}write}(type, id, v)_i$
  Precondition:
    $\neg failed \wedge replica$
    $heap \neq \emptyset$
    $clock > treat\text{-}time$
    $W = \{w \in heap : w.type = \mathsf{write}\}$
    $R = \{r \in heap : r.type = \mathsf{read}\}$
    if $W \neq \emptyset$ **then**
      $id = rqst.id = \mathsf{max}\{rqst'.id : rqst' \in W\}$
      $val = rqst.val$
      $type = \mathsf{write}$
    else
      $id = rqst.id = \mathsf{max}\{rqst'.id : rqst' \in R\}$
      $val = rqst.val$
      $type = \mathsf{read}$
    $rqst.batch \leftarrow heap \setminus \{rqst\}$
    $treat\text{-}time \leftarrow clock + treat\text{-}period$
    $treating \leftarrow treating \cup \{rqst\}$
  Effect:
    none

Input $\mathsf{read\text{-}write\text{-}ack}(id, v)$
  Effect:
    if $\neg failed \wedge replica$ **then**
      $rqst \in treating$ st. $rqst.id = id$
      $rqst.phase \leftarrow \mathsf{acknowledging}$

Output $\mathsf{snd}(\langle \mathsf{op\text{-}ack}, v\rangle)_i$
  Precondition:
    $\neg failed \wedge replica$
    $rqst \in treating$
    $rqst.phase = \mathsf{aknowledging}$
    $rqst' \in rqst.batch \vee j = rqst.rqstr$
    $rqst.batch = \emptyset \vee j = rqst'.rqstr$
    $v = rqst.val$
  Effect:
    if $rqst.batch = \emptyset$ **then**
      $rqst.phase \leftarrow \mathsf{ending}$
      $treating \leftarrow treating \setminus \{rqst\}$
    else
      $rqst.batch \leftarrow rqst.batch \setminus \{rqst'\}$

Output shrink_i
  Precondition:
    $\neg failed \wedge replica$
    $clock \geq shrink\text{-}time$
    $heap = \emptyset$
  Effect:
    if $r\text{-}access = w\text{-}access = 0$ **then**
      $replica \leftarrow \mathsf{false}$
      $shrink\text{-}time \leftarrow \infty$
    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:
    $order\text{-}expand \leftarrow \mathsf{false}$

Internal thwart_i
  Precondition:
    $\neg failed \wedge replica$
    $access\text{-}rate \geq treatment\text{-}rate$
  Effect:
    $fwd \leftarrow \mathsf{top}(heap)$
    $heap \leftarrow \mathsf{bottom}(heap)$
    $starter \leftarrow i$

Output $\mathsf{snd}(\langle fwd, str\rangle)_{i,j}$
  Precondition:
    $\neg failed \wedge replica \wedge freezing = \emptyset$
    $fwd \neq \emptyset$
    $str = starter$
    $j = \mathsf{get\text{-}thwart\text{-}nbr}$
  Effect:
    $fwd \leftarrow \emptyset$

Input $\mathsf{rcv}(\langle fwd, str\rangle)_{j,i}$
  Effect:
    if $\neg failed \wedge replica$ **then**
      if $str = i$ **then**
        $order\text{-}expand \leftarrow \mathsf{true}$
      else
        $heap \leftarrow heap \cup \{fwd\}$
        $starter \leftarrow str$

I/O Automaton 3 $Adjuster_i$ — Signature and state

Signature:
Input:
$\mathsf{shrink}_i,\ i \in I$
$\mathsf{expand}_i,\ i \in I$
$\mathsf{rcv}(m)_{j,i},\ i, j \in I, m \in M$
Internal:
$\mathsf{heal}_i,\ i \in I$
Output:
$\mathsf{snd}(m)_{i,j},\ i, j \in I, m \in M$

State:
$zones$ a set of records 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
$clock \in \mathbb{R}^{>0}$
$hb\text{-}time \in \mathbb{R}^{>0}$
$hb\text{-}period \in \mathbb{R}^{>0}$
$last\text{-}split \in I$
$a\text{-}last\text{-}split$ a mapping from I to a boolean
$detect\text{-}time$ a mapping form I to $\mathbb{R}$
$detect\text{-}period \in \mathbb{R}^{>0}$
$freezing \subseteq \{N, S, E\}$
$coeff \in \mathbb{R}^{>0}$

Transitions:
Input shrink_i
Effect:
if $\neg failed \wedge replica$ **then**
$last\text{-}split = \bot$
$replica \leftarrow \mathsf{false}$

Input $\mathsf{expand}(ra, wa)_i$
Effect:
if $\neg failed \wedge replica$ **then**
$to\text{-}add \leftarrow \mathsf{get\text{-}outside\text{-}node}()$
$replica \leftarrow \mathsf{true}$
if $((coeff \times ra) > wa)$ **then**
$\langle zones, to\text{-}add.zones\rangle \leftarrow \mathsf{get\text{-}zone}(\mathsf{hor})$
$freezing \leftarrow freezing \cup \{N\}$
else
$\langle zones, to\text{-}add.zones\rangle \leftarrow \mathsf{get\text{-}zone}(\mathsf{vert})$
$freezing \leftarrow freezing \cup \{E\}$
$index \leftarrow |nbrs| + 1$
$nbrs[index] \leftarrow to\text{-}add$
$a\text{-}last\text{-}split[to\text{-}add] \leftarrow \mathsf{true}$

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

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

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

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