HAL
open science

SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration

Emmanuelle Anceaume, Vincent Gramoli, Antonino Virgillito

To cite this version:

Emmanuelle Anceaume, Vincent Gramoli, Antonino Virgillito. SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration. [Research Report] PI 1805, 2006, pp.39. inria-00082274v1

HAL Id: inria-00082274

https://inria.hal.science/inria-00082274v1

Submitted on 27 Jun 2006 (v1), last revised 3 Jul 2006 (v2)

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° 1805

SQUARE: SCALABLE QUORUM-BASED ATOMIC MEMORY WITH LOCAL RECONFIGURATION

EMMANUELLE ANCEAUME 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

SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration

Emmanuelle Anceaume Vincent Gramoli Antonino Virgillito

Systèmes communicants
Projets ADEPT et ASAP

Publication interne n ° 1805 — June 2006 — 39 pages

Abstract: Internet applications require more and more resources to satisfy the unpredictable clients needs. Specifically, such applications must ensure quality of service despite bursts of load. Distributed dynamic self-organized systems present an inherent adaptiveness that can face unpredictable bursts of load. Nevertheless quality of service, and more particularly data consistency, remains hardly achievable in such systems since participants (i.e., *n*odes) can crash, leave, and join the system at arbitrary time. The atomic consistency guarantees that any read operation returns the last written value of a data and is generalizable to data composition. To guarantee atomicity in message-passing model, mutually intersecting sets (a.k.a. *quorums*) of nodes are used. The solution presented here, namely *Square*, provides scalability, load-balancing, fault-tolerance, and self-adaptiveness, while ensuring atomic consistency. We specify our solution, prove it correct and analyse it through simulations.

Key-words: Dynamic Distributed Systems, Self-Organization, Web Services, Quality of Service, Scalability, Performance Analysis.

(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

SQUARE: Mémoire atomique de quorums avec reconfiguration locale pour systèmes à grande échelle

Résumé : Les applications utilisées via internet nécessitent de plus en plus de ressources afin de satisfaire les besoins imprévisibles des clients. De telles applications doivent assurer une certaine qualité de service en dépit des pics de charge. Les systèmes distribués dynamiques capable de s'auto-organiser ont une capacité intrinsèque pour supporter ces pics de charge imprévisibles. Cependant, la qualité de service et plus particulièrement la cohérence des données reste très difficile à assurer dans de tels systèmes. En effet, les participants, ou *nœuds*, peuvent rejoindre, quitter le système, et tomber en panne de façon arbitraire. La cohérence atomique assure que toute lecture renvoie la dernière valeur écrite et la relation de composition la préserve. Afin de garantir l'atomicité dans un modèle à passage de message, des ensembles de nœuds s'intersectant mutuellement (les *quorums*) sont utilisés. La solution présentée ici, appelée *Square*, est exploitable à grande échelle, permet de balancer la charge, tolère les pannes et s'auto-adapte tout en assurant l'atomicité. Nous spécifions la solution, la prouvons correcte et la simulons pour en analyser les performances.

1 Introduction

Internet applications suffer from unpredictable variation of load. Such applications must provide high capacity to tolerate bursts of load implied by large scale in order to guarantee good quality of service. Typically, webservices such as *Wikipedia* [1] suffer from their popularity and must readapt their capacity to face growing interest. Moreover, high bursts of load might focus on a specific data (or *object*) in a small period of time. For instance, auctions service such as *eBay* [2] provide auctions where many participants can bid on an object during its auction lifetime. Popular objects often experience a high burst of load during the very end of their auctions. Finally, congestion and workload implied by centralized services might result in drastically increased latency and even clients requests losses.

Large scale dynamic systems have gained a widespread diffusion in recent years. Their major feature is an extreme dynamism in terms of structure, content and load. For instance, in p2p systems nodes perpetually join and leave during system's lifetime while in ad-hoc networks the inherent mobility of nodes induce a change in the size and topology of the system. These systems spontaneously organize, and desirable properties emerge from nodes collaboration [5]. This has lead to a lot of research devoted to address the issues of communication efficiency and load balancing in such systems.

High dynamism dramatically impacts on data availability [9]. Replication of data is thus necessary. Mutually intersecting sets (a.k.a. *quorums*) are a classical mean to achieve consistent data access limiting the overall replication overhead: quorums reduce the number of copies or replicas involved in reading or updating data at the same time preserving availability. Typically, Internet-scale applications such as *e-auction* or *e-booking* require data consistency and availability: auctioneers must be able to concurrently read and write their bids, while all the bids should be accessible at any time despite departures of single nodes.

Providing atomicity in distributed systems is a fundamental problem. It guarantees that despite concurrent operations invoked on a data/object, everything happens as if these operations were invoked in a sequential ordering preserving real-time precedence. In addition to this property, atomicity (a.k.a linearizability) preserves *locality* [14]. A property is local if the system as a whole satisfies this property whenever each object satisfies it. Hence, locality enables to design a concurrent application in a modular way: objects can be implemented independently from other, without requiring any additional synchronization among them to guarantee safety of the whole application. We denote by *atomic memory* of an object the set of all nodes responsible of maintaining this object. Composition of multiple atomic memories is straightforward. Atomic memory is thus a basic service that highly facilitates construction of higher services.

Finally, because of dynamism and unpredictable bursts of load, the active replicas maintaining an object value might become overloaded. For instance if the number of replicas diminishes and/or the request rate of a given object increases, then the replicas may become overloaded. Moreover, if there are too many replicas in each quorums and/or the request rate decreases, then the operation might be unnecessarily delayed. Addressing the resulting trade-off, between operation latency and capacity needed to face load requires self-adaptiveness: self-adaptiveness aims at either replicating

the object while existing replicas gets overloaded or removing replicas from quorums to minimize operation latency.

1.1 Background

Dynamic Quorums Starting with Gifford [13] and Thomas [27], quorums have been widely used in distributed systems for various applications [4, 17, 10, 26, 21]. More recently, quorums for dynamic systems appeared [14, 19, 12, 3, 24, 23, 11]. Herlihy [14] proposes quorum modification to cope with failures. In [19, 12, 11] quorum systems are subsequently replaced by independent ones to cope with permanent failures. In [3, 24, 23, 26], a quorum relies on a specific dynamic structure, and quorum probes are said *adaptive* (they contact the quorum members successively in a reactive manner). [3] proposes quorums that intersect with high probability using a dynamic De Bruijn communication graph, [24] proposes a planar overlay where a node communicates with its closest neighbor in the plane to probe a quorum, and [23] proposes a dynamic tree-structure where And/Or primitives are used to determine quorum participants when descending into the tree. In [26], the authors briefly describe strategies for the design of multi-dimensional quorum systems for read-few/write-many replica control protocols. They combine local information in order to deal with nodes dynamism and quorum sets caching in order to reduce the access latency.

Atomic Memory Emulation of shared memory in message-passing systems appeared in [7]. Since then, [19, 12, 11] have implemented atomicity in dynamic systems. These approaches provide quorum reconfiguration mechanisms to cope with permanent changes: switching from one predefined quorum configuration [12] to another or deciding upon new configurations using a quorum-based consensus [11]. These papers have a common design seed: they propose quorum system replacement, thus replacing the whole quorums participants by others. This global change requires message exchanges among all participants of previous and new configurations.

1.2 Contributions

This paper presents a *Scalable Quorum-based Atomic memory with local Reconfiguration*, namely SQUARE. *Square* is an on-demand memory ensuring *i)* atomic consistency, *ii)* fault-tolerance, *iii)* scalability, *iv)* self-adaptiveness, and *v)* load-balancing.

To provide a distributed atomic memory, *Square* replicates each atomic object at distant nodes, called *replicas*, organized into mutually intersecting sets, called *quorums*. To cope with nodes failures, the replicas of an object are organized in a logical overlay represented as a torus grid similar to the two-dimensional coordinate space of CAN [25]. This grid is divided into rectangle sub-zones where each replica is responsible of one sub-zone. Zone division occurs when replicas are added to the memory while zone merge happens after failures or leaves. Replicas responsible of zones of the same row or column form a quorum—similar quorums have been proved optimal [8, 21]. For scalability purpose, an overlay size is far lower than the system size and communication is restricted to replicas responsible of two abutting sub-zones. To provide good quality of service, the overlay self-adapts to the varying load implied by numerous clients: when the overlay gets overloaded an

active replication is triggered to diminish the global load while a low load results in reducing the overlay size thus minimizing operation latency. Finally, to balance the load, if a part of the overlay gets overloaded, then the additional load is spread among less loaded replicas.

The idea at the core of *Square* has already been published in [6]. In that work some informal building blocks for a p2p architecture supporting a self-* atomic memory called Sam are presented; *Square* proposes a specification of these building blocks and some valuable improvements for dynamic systems. Consequently, the contributions of *Square* are twofold: *i)* it specifies and simulates an algorithm supporting good properties already suggested in Sam and *ii)* proposes substantial improvements to the seminal idea. *Square* improves on the seminal idea by *i)* speeding up the operation executions, *ii)* providing atomic lock-free operations, *iii)* specifying the resulting algorithm, *iv)* proving its safety and liveness, and *v)* experimenting its behavior in a simulated large-scale dynamic system.

We prove *Square* correctness, that is, we show that read/write operation are atomic despite asynchronism, replica dynamism and replica failures. Furthermore, we show that under some reasonable assumptions, liveness is guaranteed. Finally, we show through extensive simulations, performed through a prototype implementation of *Square*, that scalability, fault-tolerance, load-balancing, and self-adaptiveness are achieved under various access requests schemes.

Road Map The paper is organized as follows. The system context and the problem of the study are presented in Section 2. The formal description of *Square* appears in Section 4. Section 5 presents proofs of safety and liveness of our algorithm. Section 6 shows the properties of *Square* by extensive simulations. In Section 7, we conclude and present some future research topics. The Appendix introduces a detailed specification using the Timed Input/Output Automaton language [20] and presents a detailed correctness proof.

2 Context and Problem

Here we present the context of the paper and the problem definition. The system consists of a set of nodes, each uniquely identified. The system is dynamic, nodes can fail and join at arbitrary time while the communication links remain reliable.

2.1 Clients

We consider a set of clients accessing a pool of shared data to consult or modify their content. In the following we use the terminology *object* for data, *read* for consult and *write* for modify. These clients can access these objects infinitely often, and concurrently. However during a finite period of time, the level of concurrency is finite. This model is often referenced in the literature as the *infinite arrival process with finite concurrency* model [22]. Each client is uniquely identified and does not necessarily know other clients. Clients can crash at any time during the execution of a read or write operation it has invoked on an object.

2.2 Atomic Objects

Each object can be accessed through read or write operations. From a client point of view, these are the only two operations that can be invoked on the object. Each accessed object is *atomic* as defined by Lynch in Lemma 13.16 of [20]. Let H be a sequence of complete invocation and response events of read and write operations, and $\prec$ be an irreflexive partial ordering of all the operations in H. Let op_1 and op_2 be two operations in H. Then

1. for any operation, there are only finitely many operations preceding it (induced by second and fourth points);
2. if the response event for operation op_1 precedes the invocation event for op_2, then it cannot be the case that $op_2 \prec op_1$;
3. if op_1 is a write and op_2 any operation then either $op_2 \prec op_1$ or $op_1 \prec op_2$, and
4. the value returned by each read operation is the value written by the last write that precedes this read according to $\prec$. (This value is an initial value, if there is no such write. A basic assumption is that objects have an initial value, so that the first read operation returns a valid value.)

This makes atomicity a very important property since despite concurrent accesses on an object, everything happens as if these operations were invoked sequentially. Another important property of atomicity is locality. A property is *local* if the system as a whole satisfies this property whenever each object satisfies it [15]. Locality is very important from both a theoretical and a practical point of view. Indeed, this property allows to design a concurrent application in a modular way: every object can be implemented independently from the others, without requiring any additional synchronization among them to guarantee the whole correctness of the application.

2.3 Atomic Memory

Atomic object defines an atomic memory abstraction. In this work, we consider that these atomic objects are handled by nodes, or servers, that may join, and leave the system infinitely often. We assume that each time a server joins, it joins with a new identity.

To face the dynamism of the environment, our atomic memory proposes two fundamental properties. First we provide *self-healing*. Self-healing is the ability of the memory to preserve persistence and availability of its objects without any external help. Practically, this is achieved by dynamically implementing each single object on several servers, and by replacing failed or left ones by new ones. In the following we use the terminology *replica* to design the set of servers that implement a single object. The second significant property of our atomic memory is *self-adaptive*. Self-adaptiveness enables the atomic memory to face the unpredictability of the environment by dynamically adapting the number of replicas to the load: the number of replicas temporarily increases during peaks of high concurrency.

The load of the memory is defined regarding to the load of each of its replica as follows: Let $\mathcal{L}_i(t)$ be the number of operations that a replica has to execute at time t. $\mathcal{L}_i(t)$ is referred in the

following as the local load of replica i at time t. Let $\mathcal{B}^i_{min}$ and $\mathcal{B}^i_{max}$ be two application dependent parameters which define respectively a lower and upper bound of the load replica i can afford. Replica $i \in I$ is *overloaded* (resp. *underloaded*) iff $\mathcal{L}_i(t) \geq \mathcal{B}^i_{max}$ (resp. $\mathcal{L}_i(t) \leq \mathcal{B}^i_{min}$). From this definition, self-adjustment guarantees that at any time, the load at any replica $\mathcal{L}_i(t)$ is such that $\mathcal{B}^i_{min} < \mathcal{L}_i(t) < \mathcal{B}^i_{max}$.

Finally, by the locality property of atomic consistency, we limit the description of *Square* to a single object. Implementation of multiple objects being identical.

3 Dynamic Horizontal/Vertical Quorum

The behavior of our atomic memory *Square* is emulated through a dynamic quorum system sampled from a dynamic but deterministic traversal. A quorum system is a set of subsets of replicas, such that every pair of subsets intersect. In the remaining we are interested in two types of quorums: horizontal and vertical ones, such that any quorum of one type simply intersects any quorum of the other type. In a dynamic setting, changes in the quorum system occur over time in an unpredictable way.

Before formally defining dynamic quorums, we briefly describe the organization of the replicas in *Square*. Replicas 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. A replica is responsible of a zone. 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 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 one replica 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] which consist in specifying 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 replica owner of the zone, and the other half is assigned to the new replica. When a replica leaves the memory (either voluntarily or because it crashes), its zone is dynamically taken over to ensure that the whole space is covered by rectangle zones, and each zone belongs to only one replica. Because of horizontal or vertical division, zones are rectangles in the plane and we refer to a zone as the product of two intervals: $I^z_{j,x} = [z.xmin, z.xmax)$ and $I^z_{j,y} = [z.ymin, z.ymax)$.

Intuitively, we define dynamic quorum sets as dynamic tiling sets, that is sets of replicas whose zones are pairwise independent and totally cover the abscissa and ordinate of the coordinate space shared by the replicas. For each real constant $c \in [0, 1)$, the horizontal tiling set $Q_{h,c}$ is made of all the replicas whose ordinate is greater than or equal to c and their ordinate is less than or equal to c. Similarly, the vertical tiling set $Q_{v,c}$ is made of all the replicas whose abscissa is greater than or equal to c and their abscissa is less than or equal to c.

Definition 3.1 (Dynamic Quorum) *Let c be a real constant with $0 \leq c < 1$.*

- *The horizontal quorum $Q_{h,c}$ is defined as the set of replicas satisfying $\{r.ymax > c \geq r.ymin\}$.*

- *The vertical quorum $Q_{v,c}$ is defined as the set of replicas satisfying $\{r \in I \mid r.xmax > c \geq r.xmin\}$.*

We define by $\mathcal{Q}_h$ and $\mathcal{Q}_v$ the set of horizontal and vertical quorums in the overlay.

Clearly, the composition of a tiling set changes over time due to replicas joins and departures, and for each real constant $c \in [0, 1)$, the horizontal tiling set $Q_{h,c}$ and the vertical tiling set $Q_{v,c}$ is defined.

Theorem 3.2 *For any horizontal quorum $Q_{h,c}$ and any vertical quorum $Q_{v,c'}$, the intersection property holds: $Q_{h,c} \cap Q_{v,c'} \neq \emptyset$.*

Proof: follows from the fact that it exists a node responsible for point (c', c) in the space.

Next, we define an ordering relation on horizontal and vertical dynamic quorums to characterize a sequence of quorums. By definition of dynamic quorums, a single dynamic quorum may correspond to different values of c. On the other hand, given a single value c, this completely characterizes a single dynamic quorum, i.e., two dynamic quorums cannot co-exist "at the same time". Thus to define the notion of sequence of quorums, we introduce the "next" relationship between quorums. Informally, a horizontal (resp. vertical) quorum Q is the next of another horizontal (resp. vertical) quorum Q', if one of Q zones has an ordinate (resp. abscissa) greater than any of Q' zones.

Definition 3.3 (Dynamic Quorum Relation) *Let $larger$ be a total ordering on horizontal quorums $\mathcal{Q}_h$ defined as: $larger(Q_{h,c}) = Q_{h,c'}$ if and only if $c > c'$. Then, $next(Q_{h,c})$ is defined as follows:*

$$\begin{cases} next(Q_{h,c}) & = Q_{h,c'} \mid c' = 0, \text{ if } larger(Q_{h,c}) = \emptyset \text{ ("borderline" of the torus)}, \\ next(Q_{h,c}) & = Q_{h,c'} \mid c' = min\{c'' > c\} \wedge Q_{h,c'} \neq Q_{h,c}, \text{ otherwise.} \end{cases}$$

Similarly we define $larger$ and $next$ on vertical quorums by replacing $Q_{h,*}$ by $Q_{v,*}$.

4 Square

Here, we present the *Square* algorithm. First, we explain the traditional idea of quorum-based read and write operations as presented in [7]. Then, we present the two protocols that are at the heart of our operation requests: the thwart providing load-balancing among the memory replicas and the traversal ensuring operation atomicity. Please recall that those algorithms result from an informal idea published in [6]. Finally we present how *Square* adapts to environmental changes, such as load and failures. For the detailed code of this algorithm, please refer to Section A.1 of the Appendix.

4.1 Read/Write Operations

The chief aim of *Square* is to provide a shared memory for dynamic environments. As said before, clients can access an atomic object of *Square* by invoking a read or a write operation on any replica this client knows in *Square*. This invocation is done through the **Operation** procedure. Pseudocode of this procedure is shown in Algorithm 1. All the information related to this request are described

in parameter $\mathcal{R}$. For instance, if the client requests a read operation then $\mathcal{R}.type$ is set to read, and value $\mathcal{R}.value$ is the default value v_0. For a write operation, $type$ is set to write and $value$ is the value to be written. The other subfields of $\mathcal{R}$ are discussed below.

Algorithm 1 Read/Write Operation

```
1:  Operation(R):
2:    if available then
3:      if overloaded then
4:        if first-time-thwart(R) then
5:          R.starter ← i
6:        Thwart(R, i)
7:      else
8:        if first-time-traversal(R) then
9:          R.initiator ← i
10:       Consult(R, i)
11:       if R.type = write then
12:         R.timestamp ←
13:           ⟨timestamp.counter + 1, i⟩
14:         Propagate(R, i)
15:         Acknowledge(R)
16:       else
17:         R.timestamp ← timestamp
18:         R.value ← value
19:         if R.value has not been propagated twice then
20:           Propagate(R, i)
21:         Return(value)
```

When such a request $\mathcal{R}$ is received by a replica, say i, i first checks whether it is currently $overloaded$ or not. Recall that a replica is overloaded if and only if it receives more requests than it can currently treat. If i is overloaded then it conveys the read/write operation request to a less loaded replica. This is accomplished by the **Thwart** process (cf. Line 6). Conversely, if i is not overloaded then the execution of the requested operation can start and i becomes the $\mathcal{R}.initiator$ of this operation. Thus, i starts the traversal process: First, i **Consults** a quorum-line to learn about the most up-to-date value of the object and an associated timestamp (Line 10). As explained later, this results in updating the local value-timestamp pair. From this point on, if the operation is a write then the counter of the request timestamp, $\mathcal{R}.timestamp$, is set to the incremented local one (Line 13) and the request timestamp identifier is set to i to break possible tie. Second, i **Propagates** in the quorum-column starting at i the new value and its associated timestamp to ensure this value will be taken into account in later operation executions. In case of write, the $\mathcal{R}.value$ propagated is the value to write, initialized by the client; while in case of a read, it is the $value$ previously consulted (Line 18). Finally, this consulted value is returned to conclude the read operation, as shown at Line 21.

Observe that, if the operation is a read and the consulted value has already been propagated twice at this replica, then the operation completes just after the **Consult** without requiring a **Propagate** phase (see Fast Read paragraph hereafter).

4.2 Traversal/Thwart Protocols

The *Square* algorithm has at its core two fundamental mechanisms, namely the traversal and thwart mechanisms. Briefly, the traversal mechanism consists in travelling the overlay vertically or horizontally in order to contact a quorum of replicas. Specifically, it ensures that a quorum is aware of the pending operation, and of the current object value. The thwart mechanism consists in traversing the overlay following a diagonal axis in order to contact at least one element of each quorum. This aims at probing quorums in order to identify the ones that are less loaded. Actually, if a non-overloaded quorum is found, then the thwart stops and the operation is executed at this quorum. If all quorums are overloaded, then the overlay is expanded. To summarize, the traversal is the elementary phase of any read/write operation, while the thwart balances the load within the memory. Both mechanisms are now detailed.

4.2.1 The Traversal Mechanism.

The Traversal, presented in Algorithm 2, consists in two procedures as shown in Figure 1(a), called respectively **Consult** and **Propagate**; the former consults the value and timestamp of a whole quorum-line whereas the latter one propagates a value and a timestamp to a whole quorum-column. Each of these procedures is executed (only if i is $available$, i.e., i is not involved in a dynamic event) from neighbor to neighbor by forwarding the information about the request $\mathcal{R}$, until both quorums (i.e., the quorum-line and quorum-column) have been traversed. The traversal ends once the initiator of the traversal receives from its neighbor the forwarding request it initially sent (i.e., the ”loop” is completed). When **Consult** or **Propagate** completes, the initiator i gets back the message (Lines 10 and 19), knowing that a whole quorum has participated. From this point on, i can continue the operation execution. That is, by directly sending the response to the requesting client if operation $\mathcal{R}$ is complete otherwise by starting a **Propagate** phase.

Remark. Note that quorums are built on the fly, meaning that they are built using an adaptive routing: when a node receives the forwarding request from its neighbor, it becomes a member of the quorum. Furthermore, elements of a quorum have only a local knowledge of the quorums they belong to. Specifically, the only members a node belonging to a quorum-line knows are its “east” and “west” neighbors. Similarly, the only members a node belonging to a quorum-column knows are its “north” and “south” neighbors. Thus the whole membership of a quorum is unknown: neither its members nor the client that invoked the operation knows it.

There are two differences between **Consult** and **Propagate**. First, the **Consult** gathers the most up-to-date value-timestamp pair of all the quorum-line replicas (Line 6) whereas the **Propagate** updates the value-timestamp pair at all replicas of the quorum-column (Line 15). Second, the **Consult**

Algorithm 2 The Traversal Protocol

1: **Prerequisite Functions:**
$next\text{-}vert\text{-}nbr()$ returns the next vertical neighbor in the sense depending on the last message receipt, to continue the propagation. If it received south-directed (resp. north-directed) message, it sends it in the south (resp. north) sense.
$other\text{-}vert\text{-}nbr()$ returns the next vertical neighbor in the opposite sense of the last message sending.
$next\text{-}hor\text{-}nbr()$ returns the next horizontal neighbor in the sense depending of the last message receipt, to continue the consultation.

2: **Consult(**$\mathcal{R}, i$**):**
3: **if** $available$ **then**
4: $\mathcal{R}.timestamp \leftarrow \mathsf{max}(timestamp,$
5: $\mathcal{R}.timestamp)$
6: $\mathcal{R}.value \leftarrow \mathsf{max}(value, \mathcal{R}.value)$
7: **if** $\neg(\mathcal{R}.initiator = i)$ **then**
8: **Consult(**$\mathcal{R}, next\text{-}hor\text{-}nbr()$**)**
9: **else if** i has already consulted **then**
10: End()

11: **Propagate(**$\mathcal{R}, i$**):**
12: **if** $available$ **then**
13: $timestamp \leftarrow \mathsf{max}(timestamp,$
14: $\mathcal{R}.timestamp)$
15: $value \leftarrow \mathsf{max}(value, \mathcal{R}.value)$
16: **if** $\neg(\mathcal{R}.initiator = i)$ **then**
17: **Propagate(**$\mathcal{R}, next\text{-}vert\text{-}nbr()$**)**
18: **else if** i has already propagated **then**
19: End()
20: **else**
21: **Propagate(**$\mathcal{R}, other\text{-}vert\text{-}nbr()$**)**

contacts each member of the quorum once following a single direction (Line 8), while the **Propagate** contacts each member of the quorum twice with messages sent in both directions (Lines 17 and 21). Consequently, if the value has been propagated twice at node i, then i knows that the value has been propagated at least once to every other replica of its quorum-column. This permits later read operation to complete without propagating this value once again.

Fast Read Operation. Not only, the traversal is lock-free compared to [6], but it does not require the confirmation phase of [12, 11], while proposing fast read operations. This results directly from the adaptiveness of our traversal mechanism. Minimizing atomic read operation latency suffers some limitations. Indeed, to guarantee atomicity two subsequent read operations must return values in a specific order. This problem has been firstly explained in [18] as the new/old inversion problem. That is, when a read operation returns value v, any later (non-concurrent) read operation must return v or

a more up-to-date value. *Square* proposes read operations that may terminate after a single phase, solving the aforementioned problem without requiring locks or additional external phase. For this purpose, the **Consult** phase of the read operation identifies if the consulted value has been propagated at enough locations. If the value v has not been propagated at all members of a quorum-column, a **Propagate** phase is required after the end of the **Consult** phase and before the read can return v, otherwise a later read might not **Consult** the value. Conversely, if a value v has been propagated at a whole quorum-column, then any later **Consult** phase will discover v or a more up-to-date value, thus the read can return v with no risk of atomicity violation.

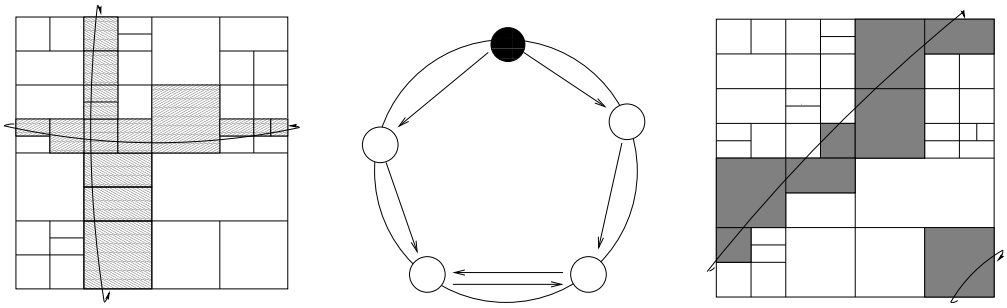

Figure 1: (a) The traversal mechanism. (b) The **Propagate** Phase. (c) The thwart mechanism.

The solution is presented in Figure 1(b) and relies on overlapping messages during the **Propagate** phase based on the fact that this phase is executed from neighbor-to-neighbor. Figure 1(b) presents a quorum-column of the torus grid as a ring where each circle models a replica and a link models a relation between two neighbors. The black circle represents the initiator of the **Propagate** phase. Unlike the **Consult** phase, the **Propagate** phase starts by two message sends: one message in each of the two senses (north and south senses in the grid). Those messages are conveyed in parallel from neighbors to neighbors until the initiator receives them back.

The idea is simple: when a replica of the ring receives a first message it simply updates their local value-timestamp pair with the one of the message, when the replica receives a second message it deduces that all the members of a quorum-column have updated its local pair to the propagated one. During a **Consult** phase of a read operation, if the (most up-to-date) consulted pair $\langle v', t' \rangle$ has been found at a replica r that has received only one message containing $\langle v', t' \rangle$, then a **Propagate** phase must occur before the end of the read operation. If replica r has received two messages propagating $\langle v', t' \rangle$, then the read can terminate immediately after the **Consult** phase. For instance, if r is one of the two bottom replicas, then the read operation can return immediately, otherwise the read must **Propagate**.

4.2.2 The Thwart Mechanism.

The Thwart, presented in Algorithm 3, relies essentially on two procedures, called **Thwart** and **Forward**. The **Thwart** is executed if i receives an operation request while it is overloaded (cf.

Algorithm 3 The Thwart Protocol

1: **Prerequisite Functions:**
$next\text{-}point\text{-}on\text{-}diagonal()$ returns the replica identifier responsible of the extreme north-east point of the zone of i.
$closest\text{-}neighbor\text{-}of(\mathcal{R}.point)$ returns the neighbor whose is responsibility is the closest to the coordinate point given as an argument.

2: **Thwart(**$\mathcal{R}, i$**):**
3: **if** $\mathcal{R}.target = i \wedge \mathcal{R}.point \in zone$ **then**
4: **if** $\mathcal{R}.starter = i$ **then**
5: **Expand()**
6: **else if** overloaded **then**
7: $\mathcal{R}.point \leftarrow next\text{-}point\text{-}on\text{-}diagonal()$
8: $j \leftarrow closest\text{-}neighbor\text{-}of(\mathcal{R}.point)$
9: **Forward(**$\mathcal{R}, j$**)**
10: **else**
11: **Operation(**$\mathcal{R}$**)**

12: **Forward(**$\mathcal{R}, i$**):**
13: **if** $\mathcal{R}.point \in zone$ **then**
14: **for** $j \in neighbors$ **do**
15: **if** $\mathcal{R}.point \in j.zone$ **then**
16: $\mathcal{R}.target \leftarrow j$
17: **Thwart(**$\mathcal{R}, \mathcal{R}.target$**)**

Line 6 of Algorithm 1). This mechanism checks the load of each quorum until it finds a non-overloaded one. For this purpose a sequence of quorum representatives, and located on the same diagonal axis, are contacted in turn, as shown in Figure 1(c). Each of these representative is a replica subsequently denoted the target of the request $\mathcal{R}.target$.

It is noteworthy that contacting subsequent replicas located on a diagonal axis leads to contacting all quorums. Furthermore, contacting only one representative per quorum is sufficient to declare that this quorum is overloaded or not. Indeed, referring to the definition of load (see Section 2), a replica becomes overloaded because of too many read/write operation requests receipt, not because of the ”load” incurred by the forwarding operation. Consequently, a quorum is not overloaded whenever its initiator is not overloaded. By definition, these replicas are not necessarily neighbors, and thus, an intermediary replica j is simply asked to **Forward** the thwart to $\mathcal{R}.target$ without checking its workload. Because of asynchrony, although a replica i sends a message to its neighbor j, at the time j receives the message j might have modified its state and might no longer be i's neighbor. (Because a new zone may have been created between nodes i and j.) To encompass this, a $\mathcal{R}.point$ indicates the final destination in the overlay coordinate space and a replica **Forwarding** or **Thwarting** first checks whether it is still responsible of this point, as expressed Lines 13 and 3.

4.3 Adapting to Environmental Changes

Here, we present self-adaptive mechanisms of *Square*. If a burst of requests occurs on the whole overlay the system needs to **Expand** by finding additional resources to satisfy the requests. Conversely, if some replicas of the overlay are rarely requested, then the overlay **Shrink**s to speed up rare operation executions. Finally, when some replicas leave the system or crash, then a **Failure-Detection** requires some of the replicas around the failure to reconfigure. Those three procedures appear in Algorithm 4.

For some reasons (e.g., failure) a replica might leave the memory without notification. Despite the fact that safety (atomicity) is still guaranteed when failures occur, it is important that the system reconfigures. To this end, we assume a periodic gossip between replicas that are direct neighbors. This gossip serves a heartbeat protocol to monitor replica vivacity. Based on this protocol, the failure detector identifies failures after a period of inactivity. When a failure occurs the system self-heals by executing the **FailureDetection** procedure: a takeover node is deterministically identified among active replicas according to their join ordering, as explained in [25]. This replica takes over the responsibility region that has been left, it reassigns a constant number of responsibility zones to make sure the responsibility-replica mapping is bijective, and it notifies its neighborhood before becoming newly $available$.

Algorithm 4 Expand and Shrink Primitives

```
1:  Expand:
2:      availabale ← false
3:      j ← FindExternalNode()
4:      ActiveReplication(j)
5:      ShareLoad(j)
6:      NotifyNeighbor(i)
7:      NotifyNeighbor(j)
8:      availabale ← true

9:  Shrink:
10:     NotifyNeighbor(i)
11:     status ← node

12: FailureDetection(j):
13:     availabale ← false
14:     TakeOver(j)
15:     NotifyNeighbor(j)
16:     availabale ← true
```

Two other procedures, namely **Expand** and **Shrink** are used to keep a desired tradeoff between load and operation complexity. When the number of replicas in the memory diminishes, fault tolerance is weakened and the overlay is more likely overloaded. Conversely, if the overlay quorum size increases, then the operation latency raises accordingly. Therefore, it is necessary to provide

adaptation primitives to maintain a desired overlay size. The **Shrink** procedure occurs when a node i is underloaded (i.e., i does not receive enough requests since a sufficiently long period of time). If this occurs, i locally decides to give its responsibility, to leave the overlay, and to become a common node (i.e., a node that does not belong to the memory). Conversely, an **Expand** procedure occurs at replica i that experienced an unsuccessful thwart. In other words, when the thwart mechanism started at i fails in finding a non-overloaded replica (i.e., the thwart turns around the memory without finding a non-overloaded replica), then i decides to expand the overlay. From this point on, initiator i becomes $unavailable$ (preventing itself from participating in traversals) chooses a common node j (i.e., a node which does not belong to *Square*) and actively replicates its timestamp and value at j. From this point on, j becomes a replica, i shares a part of its own workload and responsibility zone, and j and i notify their neighbors begin newly $available$.

5 Correctness Proof

Here, we present a sketch of the correctness proof. For the detailed proof, please refer to Section A.2 of the Appendix.

The following theorem shows the safety property (i.e., atomicity) of our system. The proof relies essentially on the fact that timestamp monotonically increases and on quorum intersection property.

Theorem 5.1 *Square implements an atomic object.*

Proof.[Sketch.] First, we show that a timestamp used in a successful operation is monotonically increased at some location. In absence of failures, it is straightforward. Assume now that replica i leaves the memory and that a replica j takes over i's zone after a **FailureDetection** event or j receives an **Expand** order: j becomes unavailable until it exchanges messages with its new neighbors (by NotifyNeighbors event), catching up with the most up-to-date value. Second we show that operation ordering implied by timestamp respects real-time precedence. A write operation **Propagates** its timestamp in any case while a read **Propagates** it if it has not been propagated yet. That is, a whole quorum-column is aware of the timestamps of ended operation. All operations contain a **Consult** phase, and by quorum intersection (cf. Theorem 3.2), discover the last timestamp. Because each written timestamp is unique and monotonically incremented, writes are totally ordered and since the value is always associated with its timestamp object specification is not violated. □

Here we show that our algorithm terminates under sufficient conditions. In order to allow the algorithm to progress, we first assume that a local perfect failure detector mechanism is available at each replica. Such a low level mechanism, available in CAN, enables a replica to determine whether one of its neighbors has failed (i.e., crashed) by periodically sending heartbeat messages to all its neighbors. Furthermore, as far as liveness is concerned, we are primarily interested in the behavior of *Square* when failures are not concentrated on a same neighborhood. This leads to the following environmental properties: *i) neighbor-failure*: between the time a replica fails and the time it is replaced, none of its neighbors fail; and *ii) failure-spacing*: there is a minimal delay between two failures occurring at the same point in the memory.

Theorem 5.2 *Eventually, every operation completes.*

Proof.[Sketch.] First, we show that a sent message is eventually received. Let i and j be two neighbors and j fails while i sends a message. Using its failure detector i will discover j's failure and a replica j' will take over j's zone. By *neighbor-failure* and *failure-spacing* assumptions, the next message from i to j' will be successfully received. Now, we show that the traversal and the thwart mechanisms terminate. We consider the worst case scenario of the thwart: the thwart wraps around the entire torus. First, observe that the overlay is a torus and the sense of subsequent messages does not change: east, north, south or diagonal. Second, by the *infinite arrival with finite concurrency* model we know that the number of **Expand** events during a finite period of time is finite. This implies that the number of replicas to contact during a traversal or a thwart is finite and both mechanisms converge successfully. □

Theorem 5.3 *Infinitely often the memory is not overloaded*

Proof.[Sketch.] By the *infinite arrival with finite concurrency* model, the level of concurrency is bounded during a period of time sufficiently long. From the above theorem, operations terminate. Thus eventually, the load on each replica i does not increase, i.e., i is not overloaded, which makes the atomic memory not overloaded by definition of the load (see Section 2). From the *infinite arrival with finite concurrency* model, these periods of time occur infinitely often. Thus infinitely often the memory is not overloaded. □

6 Simulation Study

This section presents the results of a simulation study performed through a prototype implementation of *Square*. The aim of simulations is to show *Square* properties: self-adaptiveness, scalability, load-balancing, and fault-tolerance.

The prototype is implemented on top of the Peersim simulation environment [16]. Peersim is a simulator especially suited for self-organizing large-scale system, which has proved its scalability and reliability in several simulation studies. We used the event-based simulation mode of Peersim, in order to simulate asynchronous and independent activity of nodes. The modular design of the prototype clearly separates the implementation of Square from the simulator, representing a proof of feasibility of our approach.

6.1 Environment

We simulate a peer-to-peer system containing 30,000 nodes. We recall that this is the maximum number of nodes that can be potentially added to the overlay/memory. As we show, the actual number of nodes in the memory during simulation is much lower. Here we describe the parameters of the simulator:

- We lower bound the message delay between nodes to 100 time units (i.e., simulation cycles) and we upper bound it to 200 time units.

- Any replica has to wait 1500 time units without receiving any request before deciding to leave the memory (**Shrink**).
- Each period of 2000 time units, replicas look at their buffer and treat the buffered requests, deciding to forward them (**Thwart**) or to execute them (**Traversal**).
- We send from 500 to 1000 operation requests onto the memory every 50 time units. The exact number of operation requests chosen depends on each of the following experiments.
- Each of the requested operations is a read operation with probability 0.9 and a write operation with probability 0.1.
- The request distribution can be uniform or skewed (i.e., normal). Since the results obtained with the two distributions do not present significant differences we present only those obtained with uniform distribution.
- We observe the memory evolution every period of 50 time units starting from time 0 up to $70,000$. Each curve presented below results, when unspecified, from an average measurement of 10 identically-tuned simulations.

In all experiments requests are issued at some rate during a fixed period, after which the request traffic stops. To absorb the load induced by the requests, the overlay replicates the object in various nodes of the system that are not yet in the memory. This self-adaptiveness occurs until the memory reaches a willing configuration satisfying the tradeoff between capacity and latency. We define an *acceptable configuration* as the configuration where the memory is neither overloaded, nor underloaded. This happens when some replicas of the overlay shrink while other expand. More specifically, this occurs between the first time the memory size decreases and the last time the memory size increases for a given fixed rate.

6.2 Simulation results

Here, we present the results obtained during several executions of our simulator parameterized as aforementioned.

6.2.1 Self-adaptiveness.

Figure 2 reports the number of nodes in the memory versus time. In particular, the dashed line indicates the evolution of the memory size along time, showing the adaptiveness of Square to a constant request rate. In Figure 2 the memory reaches the acceptable configuration at time 9350, while the memory leaves the acceptable configuration at time $49,200$.

Now, we focus on the three resulting periods. Before time 9350, the memory grows quickly and its growth slows down while converging to the acceptable configuration. Then, the small oscillation in the acceptable configuration is due to few nodes either leaving the memory (**Shrink**) or actively replicating (**Expand**). This means that *Square* is able to tune the capacity with respect to the request load. After time $49,200$, the memory stops growing and when the last operations are executed, load

decreases drastically causing a series of memory shrinks until one node remains. Recall that, during all three phases, although operation requests can be forwarded to other replicas, every operation is successfully executed by the memory, preserving atomicity.

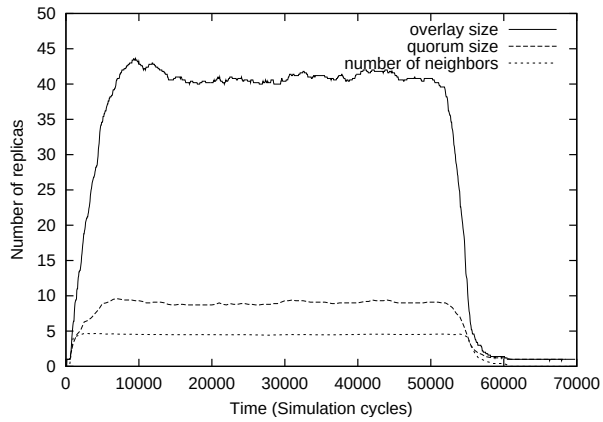

Figure 2: Evolution of memory size, mean quorum size, and mean number of neighbors by replica.

6.2.2 Scalability.

The solid line in Figure 2 plots the evolution of the average number of neighbors of each node along time and depicts an interesting result. We recall that two replicas are neighbors if they are responsible of two abutting zones. Even though the number of zones keeps evolving, the average number of neighbors per replica remains constant over time. Comparing to an optimal grid containing equally sized zones, the result obtained is similar: we can see that the number of neighbors is less than 5 while in the optimal case it would be exactly 4. We point out again that this behavior is not exclusively due to the uniform distribution of requests but it is also obtained with the normal distribution. Since only a local neighborhood of limited-size has to be maintained, the reconfiguration needed to face dynamism is scalable.

6.2.3 Load-balancing

The main contribution of the thwart mechanism is to balance the load. In order to highlight the effects of the thwart, we ran 5 different executions of the simulations, and computed the variance of the memory size. Results are reported in Figure 3. The dashed curve refers to executions where we disabled the thwart process (i.e., when a node is overloaded while it receives requests it directly expands the memory without trying to find a less-loaded replica of the memory), while the solid curve refers to executions with the thwart enabled. This simulation shows that the variance of the memory size is strongly affected by the thwart mechanism. Without the thwart, expansion might occur while a part of the memory is not overloaded, that is, the replicas become rapidly heterogeneously loaded.

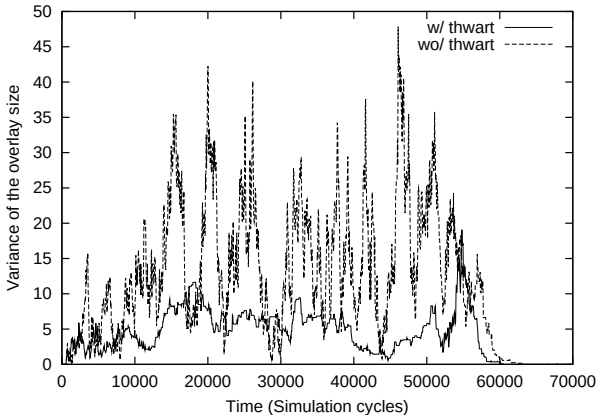

Figure 3: Impact of thwart on the variance of the memory size.

This phenomenon produces high variation in the memory size: many underloaded replicas of the memory shrink while many overloaded replicas expand. Conversely, with the thwart mechanism any replica balances the load over the memory, and verifies that the memory is globally overloaded before triggering an expansion. This makes the memory more stable.

6.2.4 Fault-tolerance.

In order to show that our system adapts well in face of crash failures, we injected two bursts of failures, while maintaining a constant request rate, and observed the reaction of the memory. Figure 4 shows the evolution of memory size as time evolves and as failures are injected. The first burst of failures occurs at the $20,000^{th}$ simulation cycle and involves 20% of the memory replicas drawn uniformly at random.

The second one occurs $20,000$ cycles later (at simulation cycle $40,000$) and involves 50% of the memory replicas. At simulation cycle $20,000$, we clearly observe that the overall number of replicas drastically diminishes. Then, few cycles later, the number of replicas starts increasing, trying to newly face the constant request rate. This phenomenon is even more important at time $40,000$ when 50% of the replicas fail. In both cases the system is able to completely return to an acceptable configuration without blocking, even after a large amount of failures has occurred.

6.2.5 Operation latency.

Experiment of Figure 5 is composed of 5 simulations with different request rates and indicates how *Square* minimizes read operation latency. First, recall that the fast-read operation contains only a **Consult** phase, thus the quorum-line size impacts more on read operation latency than quorum-column size does. We tuned *Square* such that a replica that receives more read requests than write requests tends to split horizontally its responsibility zone, when an expansion occurs. Since an op-

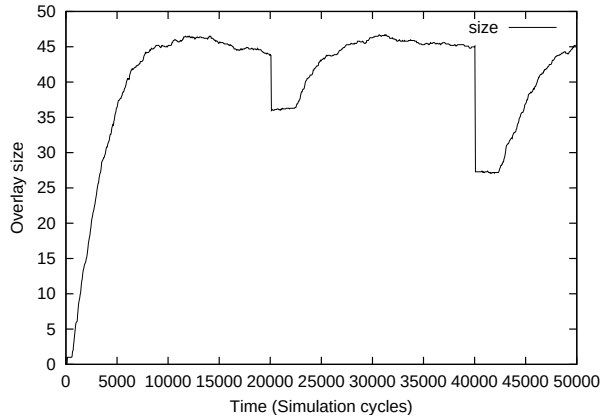

Figure 4: Self-adaptiveness in face of important failures.

eration is of type read with probability 0.9, replicas choose more frequently (in average) to split horizontally than vertically, consequently quorum-lines are smaller than quorum-columns, as depicted in the 5th and 6th columns of Figure 5. Increasing the request rate—indicated in column 1—strengthens this difference: increasing request rate enlarge the amount of operations, thus the phenomenon becomes more distinct. Furthermore, the 2nd and 3rd columns confirm our though: read operation latency is far lower than write operation latency. To conclude, even though self-adaptiveness implies that latency increases when load increases, *Square* minimizes efficiently read operation latency.

request rate	read latency (in avg)	write latency (in avg)	max. memory size	max. quorum-line size	max. quorum-column size
$1/250$	478.6	733.3	10	5	6
$1/200$	621.8	812.5	14	4	8
$1/100$	1131.8	1395.8	24	3	14
$1/50$	1500.7	2173.5	46	8	23
$1/25$	2407.9	3500.9	98	11	51

Figure 5: Trade-off between operation latency and memory size

7 Conclusion

This paper has proposed *Square* a self-adaptive atomic memory for large scale distributed systems. We have presented the two protocols that are at the heart of *Square*: the thwart protocol providing load-balancing among the memory replicas and the traversal protocol ensuring operation atomicity.

The originality of our approach is based on the self-adaptiveness of the memory to face the extreme dynamism of these systems. On the one hand, by spontaneously expanding its size when replicas become overloaded, *Square* supports bursts of load. On the other hand, by quickly shrinking to the minimal number of replicas when load decreases, *Square* minimizes operation latency. By providing fast reads, *Square* is fully adapted to applications in which consultations are more common than modifications. Despite the complexity of these systems, we have shown that atomic consistency is achievable without jeopardizing scalability, load-balancing, fault-tolerance and self-adaptiveness. Proof of feasibility of our approach has been shown through extensive simulations.

A Appendix

The solution is specified in Timed Input/Output Automaton (TIOA) language (cf. Chapter 23 of [20]) as the composition of multiple TIOA automata. This allows us to decompose the algorithm specification into several parts, each presenting a specific role of any node in the system. TIOA provides theoretical tools to formally prove correctness of algorithm. The $Load\text{-}Balancer_i$ automaton represents the module of i responsible of dealing with load burst while the $Traversal_i$ automaton represents the module responsible of overlay consistency and operation execution. Each replica i communicate through an unreliable communication channel whose automaton is not presented here, since its behavior is trivial: it receives and sends messages, and models message loss. The $Traversal_i$ automaton uses perfect $Failure\text{-}Detector_i$ and a $Takeover_i$ to detect failures and to find active replicas in the memory, respectively. These two additional automaton are not specified here; the $Failure\text{-}Detector_i$ can be seen as an oracle answering, when asked, whether a replica is failed or not; the $Takeover_i$ models the process of choosing a node among memory replicas to take over a failed zone.

Finally, $\mathcal{LBS}$, $\mathcal{TS}$, $Failure\text{-}Detector$, and $Takeover$ results from the composition of $Load\text{-}Balancer_i$, $Traversal_i$, $Failure\text{-}Detector_i$, and $Takeover_i$ for any i in I, respectively, and *Square* results from the composition of $\mathcal{LBS}$, $\mathcal{TS}$, $Failure\text{-}Detector$, $Takeover$ and the communication channel.

A.1 Detailed Specification

In the following we detail *Square* specification using the TIOA Language. First, we define several domain notations, required for the specification of *Square* modules, as it appears in Figure 6. Let I be the set of node identifiers. We refer to V, Π, and M as the sets of respectively all possible object values, operations, and messages. Finally let a *tag* be a counter, indicating the version of the object value, coupled with a node identifier to break tie, and let T be the set of tags in the system.

A.1.1 Load-Balancer

In this paper, the *Square* $Load\text{-}Balancer$ module aims simply at balancing the load among participants of the memory, a.k.a. replicas. Algorithms 5 and 6 represent the $Load\text{-}Balancer$ automaton, its signature appears from l.1 to l.16, its state appears l.17—33, and its transitions appear l.53—146.

Variables	Description
$I \subseteq \mathbb{N}$	the set of node identifiers.
V	the set of all possible values of an object.
Π	the set of all possible operations.
$T \subseteq I \times \mathbb{N}$	the set of all tags.
M	the set of all possible messages.
R	the set of all possible requests.

Figure 6: Domain

Algorithm 5 $LoadBalancer_i$ – Signature and state

Signature:
- **Input:**
  - $\mathsf{read\text{-}write\text{-}ack}(v, id)_i,\ i, id \in I, type \in \{\mathsf{read}, \mathsf{write}\}, v \in V$
  - $\mathsf{rcv}(rqst)_{j,i},\ i, j \in I$, $rqst$ a request
  - $\mathsf{fail}_i,\ i \in I$
  - $\mathsf{share\text{-}load\text{-}rcv}(b)_{j,i},\ i \in I$, b an array of requests
- **Internal:**
  - $\mathsf{load\text{-}balance}(rqst)_i,\ i \in I$, $rqst$ a request
- **Output:**
  - $\mathsf{read\text{-}write}(type, v, id)_i,\ i, id \in I, type \in \{\mathsf{read}, \mathsf{write}\}, v \in V$
  - $\mathsf{snd}(rqst)_{i,j},\ i, j \in I$, $rqst$ a request
  - $\mathsf{shrink}_i,\ i \in I$
  - $\mathsf{expand}(j)_i,\ i, j \in I$
  - $\mathsf{share\text{-}load\text{-}snd}(b)_{i,j},\ i \in I$, b an array of requests

State:
- $rqst$ a record with fields
  - $sender \in I$, the id of the requester
  - $type \in \{\mathsf{read}, \mathsf{write}\}$
  - $target \in \mathbb{R}^2$, the next requested coordinate
  - $next \in \mathbb{R}^2$, the point of the next replica (on the path to the target).
  - $str\text{-}pt \in \mathbb{R}^2$, the first requested coordinate
  - $val \in V$, the value returned by the request
- $failed$ a boolean
- $expanding$ a boolean
- $replica$ a boolean indicating whether it is a replica or not
- $batch$ the set of requests received
- $to\text{-}treat$ the set of requests that must be treated
- $treating$ the set of requests being treated
- $to\text{-}fwd$ the set of requests that must be forwarded
- $to\text{-}rspd$ the set of requests to which respond

Derived Variables:
- $overloaded = (c \leq |\{r \in to\text{-}treat \cup treating \cup batch\}|)$, where $c \in \mathbb{N}^{>0}$ is the capacity.

Initial States:
- $rqst.sender$ initialized by the requester as its own identifier
- $rqst.type$ initialized by the requester to read or write
- $rqst.target = \perp$
- $rqst.next = \perp$
- $rqst.str\text{-}pt = \perp$
- $val = v_0$ initialized as the value to write or to 0 (if the request refers to a read operation)
- $failed = \mathsf{false}$
- $expanding = \mathsf{false}$
- $replica$, true if the node maintains a value of the object, false otherwise
- $batch = \emptyset$
- $to\text{-}treat = \emptyset$
- $treating = \emptyset$
- $to\text{-}fwd = \emptyset$

State In the following we describe the state of the module, before explaining its behavior through its transitions. Observe that the main state variable is the request $rqst$. This variable is a record indicating the request id, its $sender$ (the requester), its *type*, the next $target$ed coordinates of the request, the first targeted point of the request ($str\text{-}pt$), and the value possibly returned to the $sender$ at the end of the request execution. A replica i receives some requests, before having to respond to them, to treat them, or to forward them. These request sets are denoted by respectively the $batch$, $to\text{-}rspd$, $to\text{-}treat$, and the $to\text{-}fwd$ fields. An additional $treating$ field contains operation that are currently being treated. Finally the $failed$ boolean indicates whether i is failed, the $expanding$ boolean indicates whether i is expanding, and the $overloaded$ boolean indicates whether or not the number of requests received is higher than the capacity of i.

Algorithm 6 $LoadBalancer_i$ – Transitions

```
53: Transitions:
54:   Input rcv(rqst)_{j,i}
55:     Effect:
56:       if ¬failed ∧ rqst.next ∈ zone then
57:         if rqst.str-pt ∈ zone then
58:           expanding ← true
59:           batch ← batch ∪ {rqst}
60:           last-request-time ← ∞
61:         else if rqst.target = ⊥ then
62:           rqst.str-pt ← pt|pt ∈ zone
63:           batch ← batch ∪ {rqst}
64:           last-request-time ← ∞
65:         else if rqst.target ∈ zone then
66:           batch ← batch ∪ {rqst}
67:           last-request-time ← ∞
68:         else
69:           rqst.next = closest-pt(rqst.target)
70:           to-fwd ← to-fwd ∪ {rqst}

71:   Output read-write(type, v, id)_i
72:     Precondition:
73:       ¬failed
74:       ¬expanding
75:       rqst ∈ to-treat
76:       type = rqst.type
77:       v = rqst.val
78:       id ← rqst.sender
79:     Effect:
80:       treating ← treating ∪ {rqst}
81:       to-treat ← to-treat \ {rqst}

82:   Output snd(rqst)_{i,j}
83:     Precondition:
84:       ¬failed
85:       ¬expanding
86:       (rqst ∈ fwd
87:         ∧rqst.next = closest-pt(rqst.target))
88:         ∧j = nbr(rqst.next) ∨ (rqst ∈ to-rspd
89:         ∧j = rqst.sender)
90:     Effect: none

91:   Input fail_i
92:     Effect:
93:       failed ← true

94:   time-passage(t)
95:     Precondition:
96:       if ¬failed then
97:         now + t ≤ last-request-time
98:     Effect:
99:       now ← now + t

100:  Internal load-balance(rqst)_i
101:    Precondition:
102:      ¬failed
103:      ¬expanding
104:      rqst ∈ batch
105:    Effect:
106:      if overloaded then
107:        rqst.target ← next-pt-on-diag(rqst.str-pt)
108:        to-fwd ← to-fwd ∪ {rqst}
109:      else
110:        to-treat ← to-treat ∪ {rqst}
111:      batch ← batch \ {rqst}
112:      if batch = ∅ then
113:        last-request-time ← now
114:          +unloaded-period

115:  Input read-write-ack(v, id)_i
116:    Effect:
117:      if ¬failed then
118:        if rqst ∈ treating ∧ rqst.id = id then
119:          rqst.val ← v
120:          treating ← treating \ {rqst}
121:          to-rspd ← to-rspd ∪ {rqst}

122:  Output expand(j)_i
123:    Precondition:
124:      ¬failed ∧ expanding
125:      j ← any-active-node
126:    Effect:
127:      replicating ← replicating ∪ {j}

128:  Output share-load-snd(b)_{i,j}
129:    Precondition:
130:      ¬failed ∧ expanding
131:      j ∈ replicating
132:      b ← second-half(batch)
133:    Effect:
134:      expanding ← false
135:      batch ← first-half(batch)
136:      replicating ← replicating \ {j}

137:  Input share-load-rcv(b)_{j,i}
138:    Effect:
139:      if ¬failed ∧ status = idle then
140:        batch ← b
141:        last-request-time ← ∞

142:  Output shrink_i
143:    Precondition:
144:      last-request-time ≤ now
145:      ¬failed
146:    Effect: none
```

Transitions Next, we focus on the transitions (l.53—146) of the *Load-Balancer* module. A request is received at replica i through an input rcv_i action, and its end is acknowledged by a corresponding output snd event, that might occur at a different location.

Replica i balances the load by forwarding the request to another replica in case it is overloaded. That is, the snd_i action role is twofold: either to forward a request or to respond to the requester. The choice of forwarding or treating the request received is made by i through the $\mathsf{load\text{-}balance}_i$ action. When i decides to treat a request, it adds it to its $to\text{-}treat$ set and a $\mathsf{read\text{-}write}_i$ action makes the $Traversal_i$ module treat it. When the $\mathcal{TS}$ treatment is complete, an input $\mathsf{read\text{-}write\text{-}ack}_i$ event triggered by the $\mathcal{TS}$ informs the LB module that i can respond to the requester.

The $Load\text{-}Balancer$ module chooses also to shrink and expand the memory, by removing and adding a replica, respectively. If the load is null since a sufficiently long period of time (namely the $unloaded\text{-}period$), then it shrinks, whereas if the memory is overloaded, it expands. Replica i knows that the memory is overloaded after receiving the forwarded request it sent. This receipt means that at least one replica in each quorum of the memory is overloaded. Expanding the memory through the expand_i action results in sharing the load of i with a new replica.

A.1.2 Traversal

Here we present the IOA specification of the Traversal module of *Square*. The signature and state of the corresponding $Traversal_i$ IOA are described in Algorithm 7 from line 1 to line 47. Then $Traversal_i$ specifies two modules: the operation handler and the overlay adjuster. For the sake of simplicity, first we describe states and transitions used for handling operations, then we describe the states and transitions used for adjusting the overlay.

Operation Handler of the Traversal Automaton. Each node state contains six fields: (i) the operation field op, (ii) the message field msg, (iii) the tag field tag, (iv) the value field val, (v) the $propagated$, and (vi) the $failed$ fields. The operation field op is a record containing the whole information defining an operation: its id, the id of the initiator node that trigs this operation, namely $intr$, its $type$, its current $phase$, and the $\langle tag, val\rangle$ pair indicating the state of the object from this operation standpoint. The $msg \in M$ refers to any possibly sent/received message. Next, the val field, its associated tag field, and the $propagated$ flag, all rely on an object: they express the object state from i's standpoint. The value val is its current value, the associated tag is the time-stamp or version number of this val, and the flag $propagated$ simply informs about the $\langle tag, val\rangle$ pair: whether it has been propagated or not yet. This flag has a major role since it testifies about the end of the write operation: a read can safely return a propagated tag when consulted while a non-propagated tag has to be propagated. (See the new/old inversion problem mentioned by Lamport [18].)

The behavior of replica i related to the $Traversal$ module is formally specified in Algorithms 8 and 9 (l.73–175) A read/write operation is initiated by an input $\mathsf{read\text{-}write}(*, *, id)_i$ action activated form $Load\text{-}Balancer_i$ and ends with a potentially distant $\mathsf{read\text{-}write\text{-}ack}(*, id)_*$ action. Each operation is divided into one or two phases. A write operation starts with an update (upd) phase, then comes the prop phase before completing. Unlike write, read operations can complete after a single consultation phase, namely the cons phase. However, when a write has started propagating a value but has not yet complete at a replica consulted by a read, an additional prop phase is required by the read. The $\mathsf{cons\text{-}upd\text{-}init}_i$ action initiates the first messages of the consultation and update phases initiating at replica i, while the $\mathsf{prop\text{-}init}_i$ action initiates the first propagation phase messages. Next,

Algorithm 7 $Traversal_i$ – Signature and state

1: **Signature:**
2: **Input:**
3: read-write$(type, v, id)_i,\ i, id \in I, type \in \{$read,
4: write$\}, v \in V$
5: rcv$(msg)_{j,i},\ i, j \in I, msg \in M$
6: fail$_i,\ i \in I$
7: expand$(j)_i,\ i, j \in I$
8: shrink$_i,\ i \in I$
9: failure-detect$(j)_i,\ i \in I$
10: notify-rcv$(t, v, z, n, gn)_{j,i},\ i, j \in I, t \in T, v \in V,$
11: $n \in I^*, gn \in \mathbb{N}$
12: takeover-rsp$(j, k)_i,\ i, j, k \in I$
13: replicate-rcv$_{i,i},\ i, j \in I$
14: **Output:**
15: snd$(msg)_{i,j},\ i, j \in I, msg \in M$
16: read-write-ack$(v, id)_i,\ i, id \in I, v \in V$
17: is-failed$(j)_i,\ i, j \in I$
18: notify-snd$(t, v, z, n, gn)_{i,j},\ i, j \in I, t \in T, v \in V,$
19: $n \in I^*, gn \in \mathbb{N}$
20: takeover-qry$(j)_i,\ i, j \in I$
21: replicate-snd$_{i,j},\ i, j \in I$
22: **Internal:**
23: cons-upd-init$(op)_i,\ i \in I, op \in \Pi$
24: prop-init$(op)_i,\ i \in I, op \in \Pi$
25: cons-upd-end$(op)_i,\ i \in I, op \in \Pi$
26: prop-end$(op)_i,\ i \in I, op \in \Pi$

27: **State:**
28: op an record with fields
29: $id \in \mathbb{N} \times I$, the operation id
30: $intr \in I$, the initiator replica of op
31: $type \in \{$read, write$\}$
32: $phase \in \{$idle, cons, update, prop, end$\}$
33: tag, a record with fields
34: $ct \in \mathbb{N}$, a counter
35: $id \in I \cup \{\perp\}$
36: $val \in V$
37: msg, a record with fields
38: $op \in \Pi$, the operation msg is part of
39: $sense \in \{$north, south, east$\}$, the message sense
40: $intvl \in \{$east, south, north$\} \mapsto \mathbb{R} \times \mathbb{R}$, given a sense,
41: the interval of abscissas or ordinates the message covers
42: tag, a record with fields
43: $ct \in \mathbb{N}$
44: $id \in I$
45: $val \in V$, initially v_0
46: $failed$, a boolean
47: $propagated$, a boolean
48: // The state for the adjustment follows
49: $leaving \subset I$
50: $changed \subset I$
51: $rcvd\text{-}from \subset I$
52: $nbrs \subset I$
53: $detect\text{-}time \in \mathbb{R}^{>0}$
54: $detect\text{-}period \in \mathbb{R}^{>0}$, a constant
55: $notif\text{-}time \in \mathbb{R}^{>0}$
56: $notif\text{-}period \in \mathbb{R}^{>0}$, a constant
57: $zone \in \mathbb{R}^4$, a zone
58: $nbrs$, a set of replica ids
59: $gnum \in \mathbb{N}$
60: $replica$ a record with fields
61: id, the replica id
62: $zone$, the replica zone

63: **Initial state:**
64: $op.\langle id, intr, type, phase\rangle = \langle \perp, \perp, \perp, \perp\rangle$
65: $op.tag.ct = 0$ and $op.tag.id = \perp$
66: $op.val = v_0$, the default value of the object.
67: $failed =$ false
68: $propagated =$ true
69: $leaving, changed, rcvd\text{-}from, nbrs, zones = \emptyset$
70: $clock$, the clock value at the beginning
71: $notif\text{-}time, notif\text{-}period = 0$
72: $gnum = 0$

the cons-upd-end$_i$ and prop-end$_i$ actions terminate respectively phases cons and upd, and phase prop.

A.1.3 Overlay Adjuster of the Traversal Automaton.

The behavior of replica i related to the adjuster module is formally specified in Algorithm 10. The additional states used for adjusting the memory appear in Algorithm 7 (l.49–l.62): Overlay modification impacts on replica $zone$ and set of neighbors $nbrs$. The $leaving$ set contains the replicas whose departure is known by i. After modification, affected replica are included in the $changed$ set before updating their $nbrs$ and $zone$ information. This is done by receiving newly sent information from

Algorithm 8 $Traversal_i$ – Operation transitions

```
Operation Transitions:
  Input read-write(type, v, id)_i
    Effect:
      if ¬failed ∧ status ≠ idle then
        op.type ← type
        if type = read then
          op.phase ← cons
          op.⟨tag, val⟩ ← ⟨tag, val⟩
        else if type = write then
          op.phase ← upd
          op.⟨tag, val⟩ ← ⟨⊥, v⟩
        op.intr ← i
        ops ← ops ∪ {op}

  Internal cons-upd-init(op)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      op ∈ ops
      op.phase ∈ {cons, upd}
    Effect:
      msg.op ← op
      msg.sense ← east
      msg.trajectory ← (ymax − ymin)/2
      to-send ← to-send ∪ {msg}

  Internal prop-init(op)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      op ∈ ops
      op.phase = prop
    Effect:
      msg1.op ← msg2.op ← op
      msg1.sense ← south
      msg2.sense ← north
      msg.trajectory ← (xmax − xmin)/2
      mrcv[op.id] ← 0
      to-send ← to-send ∪ {msg1, msg2}

  Output read-write-ack(v, id)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      op ∈ ops
      op.phase = end
      v = op.val
    Effect:
      op.phase ← idle

  Internal cons-upd-end(op)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      msg ∈ rcvd
      op = msg.op
      op.phase ∈ {cons, upd}
    Effect:
      op.⟨tag, val⟩ ← update(op.⟨tag, val⟩, ⟨tag, val⟩)
      rcvd ← rcvd \ {msg}
      if zone ⊂ msg.intvl[east] then
        // i has already participated
        if propagated ∧ op.type = read then
          op.phase = end
        else
          op.phase = prop
          if op.type = write then
            increments(op.tag)
      else
        msg.op ← op
        msg.sense ← east
        to-send ← to-send ∪ {msg}
      msg.intvl[east] ← msg.intvl[east] ∪ zone

  Internal prop-end(op)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      msg ∈ rcvd
      op = msg.op
      op.phase = prop
    Effect:
      ⟨tag, val⟩ ← op.⟨tag, val⟩
      if (zone ⊂ msg.intvl[north]
      ∧zone ⊂ msg.intvl[south]) then
        // i has already participated twice
        op.phase ← end
      else
        msg.intvl[msg.sense] ← msg.intvl[msg.sense]
          ∪zone
        if (zone ⊂ msg.intvl[north]
       ∧zone ⊂ msg.intvl[south]) then
          // i participates for the second time
          propagated ← true
        msg.op ← op
        to-send ← to-send ∪ {msg}
      rcvd ← rcvd \ {msg}
```

some neighbors. The set of informing neighbors is $rcvd\text{-}from$ and in order to ignore stale message from up-to-date ones, messages contain a version number called $gnum$.

The $leaving$ field represents the set of replica that as been detected as leaving by i. Observe this can be i itself if it decides to shrink. The $changed$ field contains replicas whose state might be no longer consistent. Such a replica needs to receive information from its neighbor before participating again. The set of replica from which i has already received information is denoted $rcvd\text{-}from$ and

Algorithm 9 $Traversal_i$ – Communication transitions

```
Output snd(msg)_{i,j}
  Precondition:
    ¬failed ∧ status = participating
    msg ∈ to-send
    msg.next ← next-pt-on-line(i, msg.sense,
      msg.trajectory)
    j = nbr(msg.next)
  Effect: none

Input rcv(msg)_{j,i}
  Effect:
    if ¬failed ∧ status ≠ idle ∧ msg.next ∈ zone then
      rcvd ← rcvd ∪ {msg}
      ops ← ops ∪ {msg.op}

Input fail_i
  Effect:
    failed ← true
```

$gnum$ indicates if the information received is stale or up-to-date. The notification occurs with a constant period of $notif\text{-}period$, its timeout is modeled by variable $notif\text{-}time$. The $zone$ field represents the responsibility of the replica.

Transitions. Here we describe the adjustment part. First recall that *Square* has self-adjusting capabilities, thus, it is able to expand or shrink according to some changes of its local state variables. For instance, when a replica fails the Adjuster detects it must adapt the overlay regarding to the modification. For this purpose we employ a $Failure\text{-}Detector_i$ as an external automaton. The role of this automaton at location i is simple: when a failure occurs at replica j, it informs the $Traversal_i$ of this failure location. More formally this failure detector is classified by Chandra and Toueg as eventually perfect ($\diamond P$).

Takeover. When such a failure location is found, the Adjuster looks for a replacing replica that would take over the lost zone. We use here the takeover mechanism proposed in CAN. Automaton $Takeover$ aims at finding the takeover replica. One can consider the overlay as a binary tree structure where replicas are represented by the leaves of the tree. Initially, the responsibility of the object is shared among several replicas. When an additional replica j enters the overlay, it contacts an existing replica i to take part of i's responsibility. When it occurs, the location of i in the tree structure is replaced by a virtual node whose sons are i and j, that is the entering replica and the arriving one become siblings. When a replica leaves, a replica is chosen to take its position in the tree (i.e., its responsibility). This choice is made deterministically either by taking the sibling of the departing replica (if it exists) or by making a depth first search from the father of the departing replica (descending first through the branch at the opposite side of i).

Automata Communication. More specifically, the failure detector and the load balancer at replica i trigger adjustments from location i. Automata communicate to each other through input/output actions that have the same name. When a replica fails, a $\mathsf{failure\text{-}detect}_i$ action is executed in the $Failure\text{-}Detector_i$ and $Traversal_i$ automata. Likewise, when i becomes overloaded (resp. underloaded) an expand_i (resp. shrink_i) event occurs in the $Load\text{-}Balancer_i$ and $Traversal_i$ automata. When replica i detects a failure ($\mathsf{failure\text{-}detect}_i$ occurs) or decide to leave (shrink_i occurs), then a takeover replica has to be chosen. That is i records its departure or j's failure by inserting the leav-

Algorithm 10 $Traversal_i$ – Adjustment transitions

```
Adjustment Transitions:
  Input expand(j)_i
    Effect:
      if ¬failed ∧ status ≠ idle then
        z = zone   // we choose a zone to split
        j.zone ← second-half(z)
        update(j.nbrs, ⟨j, j.zone, nbrs, 0⟩)
        zone ← first-half(z)
        update(nbrs, ⟨i, zone, nbrs, 0⟩)
        changed ← changed ∪ {j}
        rcvd-from ← ∅
        gnum ← gnum + 1
        status ← expanding

  Input shrink_i
    Effect:
      if ¬failed ∧ status ≠ idle then
        leaving ← leaving ∪ {i}

  Input failure-detect(j)_i
    Effect:
      if ¬failed ∧ status ≠ idle then
        leaving ← leaving ∪ {j}

  Input notify-rcv(t, v, z, n, gn)_{j,i}
    Effect:
      if ¬failed then
        update(⟨tag, val⟩, ⟨t, v⟩)
        update(nrbs, ⟨j, z, n, gn⟩)
        if abut( ∀k∈rcvd-from k.zones, zones) then
          // i heard from all its north and
          // south neighbors
          status ← participating
        if gn ≥ gnum then
          gnum ← gn
          rcvd-from ← rcvd-from ∪ {j}
          notif-time ← now + notif-period

  Output notify-snd(t, v, z, n, gn)_{i,j}
    Precondition:
      ¬failed ∧ status ≠ idle
      notif-time ≤ now
      t = tag
      v = val
      gn = gnum
      j ∈ nbrs
      z = j.zone
      z = j.nbrs
    Effect: none

  time-passage(t)
    Precondition:
      if ¬failed then
        now + t ≤ notif-time
        now + t ≤ detect-time
    Effect:
      now ← now + t

  Output takeover-qry(j)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      // either i is in charge of looking for
      // the takeover or it is shrinking
      j ∈ leaving ∧ i = min{k ∈ nbrs(j)}) ∨ i = j
    Effect: none

  Input takeover-rsp(j, k)_i
    Effect:
      if ¬failed ∧ status ≠ idle then
        k.zone ← j.zone
        k.nbrs ← j.nbrs
        if j = i then
          // the leaving replica is the current one
          status ← idle
        changed ← changed ∪ {k}
        leaving ← leaving \ {j}

  Output replicate-snd(t, v, z, n)_{i,j}
    Precondition:
      ¬failed ∧ status ≠ idle
      j ∈ changed
      z = j.zone
      n = j.nbrs
      ⟨t, v⟩ = ⟨tag, val⟩
    Effect: none

  Input replicate-rcv(t, v, z, n)_{j,i}
    Effect:
      if ¬failed then
        zone ← z
        tag ← t
        val ← v
        update(nrbs, ⟨j, z, n, 0⟩)
        rcvd-from ← ∅
        gnum ← gnum + 1

  Output is-failed(j)_i
    Precondition:
      ¬failed ∧ status ≠ idle
      detect-time ≤ now
      j ∈ nbrs
    Effect:
      detect-time ← now + detect-period
```

ing replica identity into the $leaving_i$ set. Let j' be the leaving replica. If i is allowed to start the takeover, takeover-qry$_i$ action query the $Takeover_i$ automaton to find a replacing replica to j'. The

corresponding response from the $Takeover_i$ contains the takeover replica identity, say k, and arrives through a $\mathsf{takeover\text{-}rsp}_i$ action. At this point, neighbors of j' have to be informed that k is taking over the responsibility of j'—those neighbors become then neighbors of k.

Expansion. When an $\mathsf{expand}(j)_i$ event occurs, an outside node j is integrated in the overlay and the responsibility zone of i is split in two. One half is removed from i's responsibility and given to j. Next, i replicates the object at j and informs j about its neighbors. This is done using the $\mathsf{replicate\text{-}snd}_{i,j}$ action at location i and the corresponding $\mathsf{replicate\text{-}rcv}_{i,j}$ action occurring at location j.

A.2 Detailed Correctness Proof

In this section, we use an assertional approach to show that the *Square* algorithm implements the solution. We show that the trace of the automata used in our *Square* composition verifies the properties mentioned above. More precisely, we show that the $Load\text{-}Balancer$ finds an underloaded quorum if such a quorum exists, while the $Traversal$ guarantees atomic consistency.

The indice $i \in I$ of any state variable represents that the variable is related to state of node i, that is $rqst_i$ is the request variable $rqst$ of node i.

A.2.1 Proof of Load-Balancing

Execution Well-Formedness. We assume that any sequence of external actions for object x is *well-formed*.

- For any $i \in I$:
  - The first event of the sequence is either a $\mathsf{rcv}(*)_i$ event or a fail_i event.
- In any well-formed execution α the following holds:
  - No fail_i event precedes any other event.
  - An input rcv_i event is immediately followed by an internal $\mathsf{load\text{-}balance}_i$ event (with no time passing).
  - An internal $\mathsf{load\text{-}balance}_i$ event putting a request in the $to\text{-}treat$ set is immediately followed by an ouput $\mathsf{read\text{-}write}_i$ event.

Notations For any $i \in I$:

- $\mathsf{closest\text{-}pt}(target \in \mathbb{R}^2)$ returns the closest point abutting i's zone and the responsible of point $target$. This is done by investigating the zone of the targeted replica and choosing a neighbor among the neighbor set according to its zone.
- $\mathsf{next\text{-}nbr}(sense \in \{\mathsf{east}, \mathsf{north}, \mathsf{south}\}, trajectory \in \mathbb{R})_i$: returns the id of the neighbor whose zone is the next in the east sense and trajectory $y = trajectory$, or in the north or south sense and trajectory $x = trajectory$ (depending on the value of argument $sense$). Note that this neighbor may be located at the opposite edge of the torus.

- next-pt-on-diag$(\mathit{str\text{-}pt} \in \mathbb{R}^2)_i$: returns the next point (out of the current replica zone) which is on the $y = x + (\mathit{str\text{-}pt}.y - \mathit{str\text{-}pt}.x)$ line and which has larger abscissa, ordinate or both than the current replica zone.
- first-half$(\mathit{batch} \subset \Pi)_i$: Let batch be an array of size s. This returns the $\lceil s/2 \rceil$ first elements of the batch array.
- second-half$(\mathit{batch} \subset \Pi)_i$: Let batch be an array of size s. This returns the $\lfloor s/2 \rfloor$ last elements of the batch array.

The proof starts with the proof of an invariant using an inductive reasoning on the length of a finite execution α. We denote the state just before and after an event by respectively s and s'.

Invariant A.1 *The size of $\mathit{to\text{-}treat}_i$ is upper bounded by the capacity c_i of replica i.*

Proof. Initially, $\mathit{to\text{-}treat}$ is empty, thus $|\mathit{to\text{-}treat}| = 0$ while $c > 0$, and the result holds.

Now assume that the property holds in some state s, that is, $|s.\mathit{to\text{-}treat}| \leq c$, we show that it holds in state s'. For this purpose we focus on the actions modifying variable $\mathit{to\text{-}treat}_i$: read-write$_i$ and load-balance$_i$. First, assume that $\mathit{read\text{-}write}_i$ occurs, that is, $|s'.\mathit{to\text{-}treat}| \leq |s.\mathit{to\text{-}treat}|$ since an element is removed from this set. Second, we focus on the load-balance$_i$ action. If $s.\mathit{overloaded}_i$ is true then $|s'.\mathit{to\text{-}treat}| = |s.\mathit{to\text{-}treat}|$. However, if it is false, $|s'.\mathit{to\text{-}treat}| = |s.\mathit{to\text{-}treat}| + 1$. Next, by definition of $\mathit{overloaded}$, we know that $|s.\mathit{to\text{-}treat}| < c$. Combining this inequation with the previous equation leads to the result: $|s'.\mathit{to\text{-}treat}| \leq c$. □

The following Lemma shows that if request is forwarded from i and it is received by j such that j considers it, then j belongs to the next horizontal quorum and to the next vertical quorum of i. This Lemma is necessary to show that the look-up goes through all dynamic quorums $\forall c$, $0 \leq c < 1$, $Q_{h,c}$ and $\forall c'$, $0 \leq c' < 1$, $Q_{v,c'}$.

Lemma A.2 *If $r.\mathit{target}_i \in i.\mathit{zone}$ or $r.\mathit{target}_i = \perp$ and $\mathit{overloaded}_i$ is* true *when a $\mathit{load\text{-}balance}(r)_i$ occurs, then the next* rcv(r) *of α occurs at location j with $r.\mathit{next} \in j.\mathit{zone}$ and $r.\mathit{target} \in j.\mathit{zone}$ is such that $\exists c : i \in Q_{h,c} \wedge j \in \mathit{next}(Q_{h,c})$ and $\exists c' : i \in Q_{v,c'} \wedge j \in \mathit{next}(Q_{v,c'})$.*

Proof. Assume that $r.\mathit{target}_i \in i.\mathit{zone}$, $\mathit{overloaded}_i =$ true and $\mathit{load\text{-}balance}(r)_i$ occurs. Assume also that the next rcv(r) event of α occurs at j with $r.\mathit{next} \in j.\mathit{zone}$ and $r.\mathit{target} \in j.\mathit{zone}$.

Assuming this, the snd$(r)_i$ event occurring at i forwards $r \in \mathit{to\text{-}fwd}_i$ with $r.\mathit{target}_i =$ next-pt-on-diag$(r.\mathit{str\text{-}pt}_i)$. By definition of the next-pt-on-diag function, we know that $r.\mathit{target}$ is the point $(\mathit{zone.xmax}_i, \mathit{zone.ymax}_i)$.

Assume, by absurd, that there is no c verifying $i \in Q_{h,c} \wedge j \in \mathit{next}(Q_{h,c})$. Let $Q_{h,c'}$ be $\mathit{next}(Q_{h,c})$, that is, $c' \neq \mathit{min}\{c'' > c\} : Q_{h,c'} \neq Q_{h,c}$. Now assume $c' > c$ is not the minimum such that $Q_{h,c'} \neq Q_{h,c}$, then $\exists c'''$, $c < c''' < c'$ such that $Q_{h,c'''} \neq Q_{h,c}$. By examination of the rcv$_{*,j}$ action, $r.\mathit{next} \notin \mathit{zone}_j$ contradicting assumptions, thus, $\exists c : i \in Q_{h,c} \wedge j \in \mathit{next}(Q_{h,c})$. The proof for the vertical quorums is similar. □

The following Theorem and corollary shows that $\mathit{Load\text{-}Balancer}$ balances the load.

Theorem A.3 *If β is a trace of $\mathcal{LBS}$ that satisfies the $\mathcal{LBS}$ environment assumptions, then β satisfies the following conditions:*

1. *If i is overloaded when a request (whose target is i) is received, then the request is forwarded.*
2. *If i is not overloaded when a request (whose target is i) is received, then the request is treated.*
3. *If a request is forwarded, then it is forwarded to (at least) the next quorums.*

Proof. We show each of the three properties separately. For Property (1), we show that request is added to the $to\text{-}fwd$ set, for Property (2) we show the request is added to the $to\text{-}treat$ set. For Property (3), we prove that the target is set to a replica belonging to the next quorums and that the request is forwarded to an intermediary replica putting it in its $to\text{-}fwd$ set. Finally by well-formedness assumptions, we know that treatment or forward eventually occurs if it is decided locally, thus the three properties hold.

1. To ensure Property (1), an overloaded replica receiving a request must forward it. That is assume that client j sends a request to the memory and the corresponding $\mathsf{rcv}(rqst)_{j,i}$ event occurs at replica i while i is overloaded. Observe that if the request target is defined ($\neq \perp$) and is not i, then i simply forward the request instantaneously: $rqst$ is added to the $to\text{-}fwd$ set. Now, assume that $rqst$ is added to the $batch$ set. By well-formedness condition a $\mathsf{load\text{-}balance}(rqst)_i$ immediately occurs. Since no interleaving action occur, the $treating$ and $to\text{-}treat$ sets remain unchanged. That is, i is still overloaded, thus Invariant A.1 implies that $rqst$ is added to the $to\text{-}fwd$ set.
2. For Property (2) the proof is similar to the one mention above. We ignore the case where $rqst$ is added to the $to\text{-}fwd$ set during the $\mathsf{rcv}(rqst)_{i,*}$ event. Thus, assume that $rqst \in batch$. By well-formedness conditions and examination of the $\mathsf{load\text{-}balance}(rqst)_i$ action $rqst$ is added to the $to\text{-}treat$ action.
3. For Property (3), assume that the request $rqst$ is forwarded. Request $rqst$ is added to the $to\text{-}fwd$ set either by a $\mathsf{rcv}(rqst)_i$ action or by a $\mathsf{load\text{-}balance}(rqst)_i$ action where $overloaded_i = \mathsf{true}$.
   - $\mathsf{rcv}(rqst)$: when this action occurs the $rqst.target$ is unchanged and the $rqst.next$ is reset to the closest point to the target. That is, the receiver of the request has been set during an earlier event.
   - $\mathsf{load\text{-}balance}(rqst)$: when this action occurs, a $\mathsf{rcv}(rqst)$ event has previously occurred where $rqst.target \in i.zone$ or $rqst.target = \perp$. By Lemma A.2, we know that the $rqst$ is forwarded to a replica j belonging to next quorums.

By well-formedness assumptions, requests of the $to\text{-}treat$ set are effectively treated while the requests of the $to\text{-}fwd$ set are effectively forwarded. Consequently, the three properties hold and an execution β satisfying $\mathcal{LBS}$ assumptions, satisfies the load-balancing condition. □

Next, we define the quorum underload and quorum overload.

Definition A.4 (Quorum Overload/Underload) *Let $Q \subset I$ be a quorum. Q is* overloaded *(resp.* underloaded*) if the node $i \in Q$ contacted during the thwart is overloaded (resp. underloaded).*

Corollary A.5 *Let $P = \{S \subset I\}$ be a set of specific subsets of replicas in the memory. Assume the look-up procedures terminates. It exists a non-overloaded set $S \in P$ in the memory, then the look-up procedure returns an element of it.*

Proof. The proof is based on the results of A.3. First-of-all, we consider that the quorum(s) returned is(are) the quorum(s) where the operation is executed. By assumption we know that the number of quorums in the memory is finite. Observe that there is a total order on quorum sets in the memory (cf. Definition 3.3). That is and by Propositions (1) and (3), we know that there are two cases: either (i) the procedure does not complete, or (ii) if an overloaded replica is encountered, then the next quorum set is contacted. That is eventually, all quorums are contacted. Note that if the contacted replica is overloaded, then its quorums (the quorums this replica belongs to) are also overloaded (cf. Definition A.4). Finally, by Proposition (2) if a contacted replica is not overloaded, then it treats the request, thus the procedure returns the quorum it belongs to. □

A.2.2 Proof of Atomicity

The complete implementation of the $\mathcal{TS}$ System, namely $\mathcal{TS}$, is given by the composition of the $Traversal_i$, the $Takeover_i$ and the $Failure\text{-}Detector_i$ automata for all i, with the communication channels.

Execution Well-Formedness. We assume that any sequence of external actions for object x is *well-formed*.

- For any $i \in I$:
  - The first event of the sequence is either a $\mathsf{read\text{-}write}_i$ event, a $\mathsf{replicate\text{-}rcv}_i$ event, or a fail_i event.
  - No fail_i event precedes any other events.
- In any well-formed execution α the following holds:
  - At most one $\mathsf{read\text{-}write}(*, *, id)_*$ event occurs. That is, an operation is uniquely identified, whatever the location the operation is requested.
  - Every $\mathsf{read\text{-}write}(*, *, id)_*$ event has a corresponding $\mathsf{read\text{-}write\text{-}ack}(*, *, id)_*$ event following it.

Notations For any $i \in I$:

- *tag ordering relation*: Let $>_t$, be an ordering relation on T such that for all $t_1 \in \mathbb{N} \times I$, $t_2 \in \mathbb{N} \times I$, $t_1 >_T t_2$ if one of the following conditions hold:
  - $t_1.counter > t_2.counter$
  - $t_1.counter = t_2.counter \wedge t_1.id > t_2.id$
- $\mathsf{update}(nbrs \subset I, \langle j \in I, z \in \mathbb{R}^4, n \subset I, gn \in \mathbb{N}\rangle)_i$: updates the neighbors array of replica i using j's zone, z, and j's neighbors, n. This update is done regarding to these information timestamp (gn).
- $\mathsf{update}(\langle tag \in \mathbb{N} \times I, val \in V\rangle, \langle t \in \mathbb{N} \times I, v \in V\rangle)_i$: updates the local pair $\langle tag, val\rangle$ of i with $\langle t, v\rangle$. If $t >_T tag$ then $\langle tag, val\rangle \leftarrow \langle t, v\rangle$, else nothing happens.

- $\mathsf{nbr}(i \in I, sense \in \{\mathsf{east}, \mathsf{north}, \mathsf{south}\})_i$: returns a replica identifier of i's neighbors whose zone abuts $sense$ edge of a zone of i.
- $\mathsf{abut}(\{pt \in \mathbb{R}\}, \{z \in \mathbb{R}^4\})_i$: returns a boolean indicating whether or not the set of points pt abuts the lower or upper edge of each zone z.
- $\mathsf{increments}(tag \in \mathbb{N} \times I)$: modifies tag such that its counter subfield is incremented. For instance let $tag = \langle ct, i \rangle$, the function modifies it such that $tag = \langle ct + 1, i \rangle$.

Invariant Assertions The following invariant shows that for a given replica, its tag never decreases. This is due to the fact that during a propagation phase, the tag is modified only if the propagated tag is larger than the local one.

Invariant A.6 *Local tag is larger than any other tags encountered so far. In other words, every replica tag is monotonically increasing.*

Proof. First, we show that any tag is unique. Observe that any tag is associated with a local counter such that every new tag gets assigned a different value. In order to break tie among distant node, a tag gets assigned the identifier of the node creating it as a low-weight number.

Then, we focus on actions modifying the state of the tag variable: $\mathsf{replicate\text{-}rcv}_i$, $\mathsf{notify\text{-}rcv}_i$, and $\mathsf{prop\text{-}end}_i$. Each of these operations updates tag and $value$. The $\mathsf{update}(p_1, p_2)$ function sets the local p_1 tag-value pair to p_2 only if $p_2.tag \geq p_1.tag$ tag (i.e., only if p_2 is more up-to-date than p_1). As a result, p_1 value might increase but can not decrease. □

Adjustment Guarantees Next, we show that despite adjustment transitions, the tag that has been propagated at some place is non-decreasing. The core of this proof relies on the adjustment modifications. By Invariant A.6, we know the tag is non-decreasing locally. However, if the replica leaves or expands the memory, it is not straightforward that the new responsible of the same coordinate gets not a stale tag-value pair (i.e., with a lower tag).

For instance, assume that a propagation targets a point pt, thus, the tag of the responsible of pt is set to t_1. Next, suppose that a subsequent propagation occurring through points close to pt changes its responsible tag to t_2. If this replica expands, then it is possible that the entering replica becoming responsible of pt might contact all its north and south neighbors learning about $t_0 \neq t_2$ such that $t_0 < t_1$. In this case, a tag at a location might decrease. Next, we show that the tag freshly propagated at some point is not decreasing at this point. In the following we say a replica is part of memory, if and only if its status is $\mathsf{participating}$.

Lemma A.7 *Let S be the set of points targeted by the last completing propagation. The tag of the replicas responsible for S is non-decreasing.*

Proof. First-of-all by Invariant A.6, for a specified replica its tag is monotonically increasing.

Initially any node $status$ is idle and when an expand_i event occurs $status$ is set to idle. In order for the replica to be part of the memory, a $notify\text{-}rcv_{j,i}$ must occur, where $\mathsf{abut}(\bigcup_{\forall k \in rcvd\text{-}from} k.zones, zones)$. This means that the north or south edge of the $zones$ are entirely covered by the zones of all answering neighbors k. In order for a replica to be added to

the $rcvd\text{-}from$ set, the message received when the rcv occurs, must contain an up-to-date gossip number gn, i.e., a gn number at least as large as the local $gnum$ number that has been previously sent. The $gnum$ value is updated if a larger one is received or it is incremented during an expand or a replicate-rcv event. These two last events are obviously followed by a $notify\text{-}snd_{i,j}$ containing the new gossip number $gnum$ and the corresponding reply $notify\text{-}snd_{j,i}$ before $status$ becomes participating. □

Phase Guarantees Next we show, that if the consulted tag is fully propagated, then the sequence of consulted tag is non-decreasing.

Lemma A.8 *Let ϕ_1 be a propagation phase such that $tag(\phi_1)$ has been fully propagated. If a phase ϕ_2 starts after ϕ_1 ends, then $tag(\phi_2) \geq tag(\phi_1)$.*

Proof. If the tag is fully propagated, then Lemma A.7 implies that this tag is non-decreasing at the point targeted by the propagation despite dynamism. The proof aims at showing that there exists such a point consulted by ϕ_2.

For any msg_* of a phase, its $trajectory$ and sense $sense$ are unchanged during the whole phase execution. This comes directly from the fact that cons-upd-init or the prop-init occurs once in any phase. When a message is sent from i to j (a $\mathsf{snd}_{i,j}$ event occurs), the receiver is chosen among neighbors of i such that the zone of j abuts the zone of i at the intersection between $sense$ edge and the $trajectory$ line. Now assume that a message is sent from i to j, that is, no expansion is pending since $status$ must be participating. Furthermore, i did not leave the system yet (neither by failing nor by shrinking) for the same reason. Because of this $status$ field, either a node is not participating or its neighbors are up-to-date. By examination of the $\mathsf{cons\text{-}upd\text{-}end}_i$ (or $\mathsf{prop\text{-}end}_i$) action, we know that the phase completes when its msg reaches back the starting point after wrapping around the torus grid. By examination of the code, when a $\mathsf{rcv}_{i,j}$ event occurs, observe that either j's zone abuts the one i had when the message has been sent, or $rqst.next \notin zone_j$, and nothing happens during this event. That is, when the consultation/update (resp. propagation) phase ends, replicas of all zones located on the line of equation $y = trajectory$ (resp. $x = trajectory$) have been contacted. Because of the column-line intersection and Lemma A.7, the tag consulted by ϕ_2 is at least as large as the tag propagated during phase ϕ_1. □

In the main theorem, we show that the tag orders the operations as mentioned in Section 2.2.

Theorem A.9 *If β is a trace of $\mathcal{TS}$ that satisfies the $\mathcal{TS}$ assumptions, then β satisfies the atomicity definition.*

Proof. We show each property separately.

For property (2), assume first that there is a propagation phase in operation π_1. If the response point of operation π_1 precedes the invocation point of operation π_2 then propagation phase ϕ_1 of π_1 completes before consultation phase ϕ_2 of π_2 starts. That is, Lemma A.8 implies that $tag(\phi_1) \leq tag(\phi_2)$. Now assume that there is no propagation phase in π_1. Because of operation termination, this implies π_1 is a read operation and the consulted tag-value pair is propagated (cf. l.128 and 127 of Alg.8), indicating it exists a propagation ϕ_p that has successfully completed earlier

such that $tag(\phi_1) = tag(\phi_p)$. Since ϕ_1 response precedes ϕ_2 invocation, ϕ_p response precedes also ϕ_2 invocation and Lemma A.8 implies that $tag(\phi_p) = tag(\phi_1) \leq tag(\phi_2)$.

Property (3) follows from the fact that the tag of a write operation is incremented. By examination of the cons-upd-end if the consultation phase ends while the operation is a write, then the tag is incremented.

Property (4) observe that the value returned during a read operation π is the one associated with $tag(\pi)$. That is, the result is straightforward from Properties (2) and (3).

For Property (1), observe that any response of operation π_1 must be preceded by a finite number of invocations. Let Π' be the finite set of operations corresponding to such invocations. By Property (2) there is no operation $\pi_2 \notin \Pi'$ such that $\pi_2 \prec \pi_1$. □

A.2.3 Proof of Liveness

In this section we prove that *Square* terminates. That is, we give assumptions ensuring that the algorithm terminates. In other words we prove that operations of the $\mathcal{TS}$ automaton and the look-up process of the $\mathcal{LBS}$ eventually completes.

The goal is to show that the routing process—at the core of operation and look-up—converges. For this purpose, we assume that the system eventually stabilizes and message delays becomes bounded.

Assumptions. In the following we consider the global automaton $\mathcal{S}quare$ as the composition of $\mathcal{LBS}$ and $\mathcal{TS}$. Let α be an execution of the global system $\mathcal{S}quare$, and let α' be a finite prefix of α where the system can be unstable. We refer to $\ell time(\alpha')$ as the time when the last event of α' occurs and when the system stabilizes.

While information is routed among replicas, coordinates are used to make sure the receiver corresponds to the up-to-date target. We refer to pt_0 as the current point reached by the routing mechanism. Assume pt_1 is the next point to contact during the routing and pt_n is the target point where the routing ends.

We need to make two lists of assumptions about (i) communication constraint, (ii) dynamism constraint inside the memory.Preliminary, we define timing bounds on some elementary procedure. We assume that the time required for a failure to be detected is upper bounded by $fd\text{-}time$. Moreover, we upper bound the time of a local reconfiguration including the takeover mechanism by $rf\text{-}time$. Finally, we upper bound the time for an expansion to complete by $exp\text{-}time$.

We give some assumptions on communication link.

1. *communication-bound*: If a $\mathsf{snd}_{i,j}$ event occurs at time d and j is active at time $t + d$, then a corresponding $\mathsf{rcv}_{j,i}$ occurs at time $t + d$.
2. *communication-frequency*: We assume that when $\mathsf{snd}_{i,*}$ and $\mathsf{notify\text{-}snd}_{i,*}$ preconditions are satisfied, these events occur with a d frequency. If a $\mathsf{notify\text{-}rcv}_{*,i}$ event occurs, then a $\mathsf{snd}_{i,*}$ and a $\mathsf{notify\text{-}snd}_{i,*}$ occurs immediately.

Note that property (1) is important since it implies that the overlay is never partitioned but remains connected. We assume several constraints of the replica dynamism in the memory. Assuming that replica failures and replica expansions are finite, routing among nodes becomes possible.

1. *neighbor-failure*: Between the time a replica fails and the time it is replaced, at least all zones abutting a vertical edge of the current one have active responsible.
2. *failure-spacing*: There is at least $\mathit{fd\text{-}time} + \mathit{rf\text{-}time} + \mathit{exp\text{-}time} + 2d + \epsilon$ between two subsequent fail event occurring at replicas responsible of pt_1 point and no failure occurs at the current replica (the one responsible of pt_0).

First we show that the algorithm of the $\mathcal{L}BS$ terminates, that is, when the requested replica is overloaded, then the propagated request is eventually treated by some replica of the memory. Second, we show that an operation resulting from a treated request eventually completes. Lastly, we conclude that any request invoked by a client completes.

The following lemma states that the routing progresses. It shows that if a $\mathsf{snd}_{i,*}$ occurs while node i does not fail during sufficient amount of time ($max(t + \ell time(\alpha')) + \mathit{fd\text{-}time} + \mathit{rf\text{-}time} + \mathit{exp\text{-}time} + d$), then a corresponding $\mathsf{rcv}_{*,j}$ occurs.

Lemma A.10 *If a* snd *event occurs at time t then a* rcv *event occurs before time $max(t + \ell time(\alpha')) + \mathit{fd\text{-}time} + \mathit{rf\text{-}time} + \mathit{exp\text{-}time} + 2d$.*

Proof. At time $t' = max(t, \ell time(\alpha'))$, the system is stable and snd_i occurs. Let i and j be the replicas responsible of pt_0 and pt_1, respectively, at time t'. Assume a fail_j event occurs at time t'. That is no later than time $t'' = t' + \mathit{fd\text{-}time} + \mathit{rf\text{-}time}$ another replica has taken over j's responsibility. Let j' be this replica. By *failure-spacing* assumption, we know that no *fail* event can occur at i or j' until time $t'' + \mathit{exp\text{-}time} + 2d + \epsilon$. Because of *communication-bound* assumption, at time $t'' + d$, $\mathsf{notify\text{-}rcv}_{*,i}$ occurs and i neighbors set becomes up-to-date. Since the $\mathsf{snd}_{i,j'}$ action was enable at time t' and an expansion lasts less than $\mathit{exp\text{-}time}$, message is sent at time $t'' + \mathit{exp\text{-}time} + d$, then the $\mathsf{rcv}_{i,j'}$ event occurs at time $t'' + \mathit{exp\text{-}time} + 2d$. □

Here we show that a message of the $\mathcal{L}BS$ can not be forwarded infinitely often.

Lemma A.11 *If a* $\mathsf{snd}(r)_*$ *event occurs in any $\mathcal{L}BS$ execution, then eventually either r is put in $\mathit{to\text{-}treat}$ or an* expand *event occurs.*

Proof. For this proof observe that messages relying on the same request r follow the same trajectory. The starting point, $r.\mathit{str\text{-}pt}$ is set when the request is received for the first time from the client (i.e., when $r.\mathit{target} = \perp$) at i. Because of *communication-bound* assumption, this information is conveyed through every messages of the $\mathcal{L}BS$ automaton, and no replica will ever modify it. That is, $r.\mathit{target}$ is set once and remains unchanged thereafter.

Forwarded messages target the responsible of point $\mathsf{next\text{-}pt\text{-}on\text{-}diag}(r.\mathit{str\text{-}pt})$. In order to reach this target, messages go through the replica responsible of the $\mathsf{closest\text{-}pt}$ to the target. Observe that $\mathsf{next\text{-}pt}(r.\mathit{str\text{-}pt})$ is chosen as the next point on axis of equation $y = x + r.\mathit{str\text{-}pt}.y - r.\mathit{str\text{-}pt}.x$ and Lemma A.10 implies that a message sent is eventually received. Observe that the *i*nfinite arrival

process with finite concurrency model prevents the number of expand from being infinite during a closed interval of time. Since the overlay is a torus and te finite number of expansions implies that the number of replicas to contact is finite, a never treated request is forwarded back to $r.str\text{-}pt$. If this occurs, the rcv event implies an expansion. □

Next Theorem shows that the look-up procedure, which is at the core of the $\mathcal{L}BS$ automaton, terminates. Hence, if a request r is forwarded in $\mathcal{L}BS$, r is eventually treated in $\mathcal{T}S$.

Theorem A.12 *If a* $\mathsf{snd}(r)_{i,*}$ *occurs where request* $r \in$ *to-fwd then eventually a* $\mathsf{read\text{-}write}(*, *, id)$ *occurs with* $r \in$ *to-treat, and* $id = r.sender$.

Proof. First, we show that either a replica treats the request it receives or forwards it. By well-formedness assumptions, we know a request is eventually treated and forwarded when put in the *to-treat* and *to-fwd* sets, respectively. The $\mathsf{rcv}(r)$ action put it in the *batch* or the *to-fwd* set if i is not *failed*. Assume that $r \in batch$ just after this event occurs. Then the $\mathsf{load\text{-}balance}_i$ action preconditions are satisfied. As a result of this action execution, $r \in$ *to-treat* $\cup$ *to-fwd* leading to the result. By Lemma A.11, we know that a request can not be infinitely forwarded. It follows that the process completes. □

In the following Theorem, we show that the operation execution, which is at the core of the $\mathcal{T}S$ automaton, terminates.

Theorem A.13 *If a* $\mathsf{read\text{-}write}(*, *, id)$ *input event occurs, then eventually the corresponding* $\mathsf{read\text{-}write\text{-}ack}(*, *, id)$ *output event occurs.*

Proof. First we focus on showing that a consultation phase terminates. At the beginning of the consultation or update phases the *trajectory* is set to a horizontal axis and sense is set to east sense. Observe then that the trajectory and sense of messages never change within the considered phase. By Lemma A.10 and the finite expansions induced by our model, we know that the routing converges, thus, the starting point of the phase is reached back. When the cons-upd-end event occurs at the replica responsible of the starting point, $zone \in msg.intrvl[\mathsf{east}]$ and the phase ends. Second we focus on the propagation phase. The proof is similar, though messages are sent in two opposite senses namely north and south. That is the fix-point is not reached until both $zone \in msg.intrvl[\mathsf{north}]$ and $zone \in msg.intrvl[\mathsf{south}]$, meaning both type of messages wrapped around the torus and went back to the starting point. When a prop-end occurs while this is satisfied, the phase ends. □

Corollary A.14 *The system verifies the two following properties:*

1. *Eventually every operation completes,*
2. *Eventually the system gets non-overloaded.*

Proof. Property (1) follows directly from Theorem A.12 while the property (2) follows directly from Theorem A.13. That is, *Square* terminates. □

References

[1] eBay. http://www.ebay.com/.

[2] Wikipedia. http://www.wikipedia.org/.

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

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

[5] E. Anceaume, X. Defago, M. Gradinariu, and M. Roy. Towards a theory of self-organization. In *Proc. of 9th Intl Conference on Principles of Distributed Systems*. Springer-Verlag, 2005.

[6] E. Anceaume, M. Gradinariu, V. Gramoli, and A. Virgillito. P2P architecture for self* atomic memory. In *Proc. of 8th Intl Symposium on Parallel Architectures, Algorithms and Networks*, pages 214–219, Dec. 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] B. Awerbuch and P. Vitanyi. Atomic shared register access by asynchronous hardware. In *Proc. of 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[9] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[10] 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.

[11] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *Proc. of 9th Intl. Conference on Principles of Distributed Systems*. Springer-Verlag, Dec. 2005.

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

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

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

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[16] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, number 2977, pages 265–282. Springer-Verlag, April 2004.

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

[18] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

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

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

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

[22] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proc. of the 14th Intl Conference on Distributed Computing*, pages 164–178, London, UK, 2000. Springer-Verlag.

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

[24] M. Naor and U. Wieder. Scalable and dynamic quorum systems. In *Proc. of the 22th annual symposium on Principles of distributed computing*, 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] 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.

[27] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.