

SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration

Vincent Gramoli, Emmanuelle Anceaume, Antonino Virgillito

▶ **To cite this version:**

Vincent Gramoli, Emmanuelle Anceaume, Antonino Virgillito. SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration. ACM Symposium on Applied Computing (SAC), 2007, France. pp.12. hal-00916758

HAL Id: hal-00916758
https://hal.science/hal-00916758

Submitted on 13 Dec 2013

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.

SQUARE: Scalable Quorum-Based Atomic Memory with Local Reconfiguration

Vincent Gramoli
IRISA, Université de Rennes 1
35042 Rennes, France
vgramoli@irisa.fr

Emmanuelle Anceaume
IRISA, CNRS
35042 Rennes, France
anceaume@irisa.fr

Antonino Virgillito
ISTAT
00184 Roma, Italy
virgilli@istat.it

ABSTRACT

Internet-scale 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., *nodes*) can crash, leave, and join the system at arbitrary time. Atomic consistency guarantees that any read operation returns the last written value of a data and is generalizable to data composition. To guarantee atomic consistency in message-passing model, mutually intersecting sets (a.k.a. *quorums*) of nodes are used. The solution presented here, namely *Square*, uses self-adaptiveness and load-balancing to provide atomic consistency in large-scale dynamic distributed systems. This paper presents the *Square* algorithm and uses extensive simulation to show it achieves its desirable properties.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Simulation, Reliability.

Keywords

Dynamic Distributed Systems, Adaptiveness, Scalability, Consistency, Performance Analysis.

1. INTRODUCTION

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 peer-to-peer (p2p) systems nodes perpetually join and leave during system lifetime while in ad-hoc networks the inherent mobility of nodes induce a change in the system topology. Internet applications suffer from unpredictable variation of load. Typically, 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* [1] 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 applied to centralized services might result in drastically increased latency and even losses of clients requests.

To increase availability, replication of data is necessary. Mutually intersecting sets of data replicas (a.k.a. *quorums*) are a classical mean to achieve consistent data access limiting the overall replication overhead. Atomic consistency 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. Atomicity (a.k.a linearizability) preserves an important property, called *locality* [10], often considered as the capability of being compositional. However achieving atomic consistency in face of high dynamism requires additional healing mechanisms to guarantee object persistence despite accumulated failures. This type of improvement leads naturally to *dynamic quorums*.

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 applied to a given object increases, then the replicas may become overloaded. Moreover, if there are too many replicas in each quorum 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.

Contributions. This paper presents the experimental analysis of 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)* load-balancing, and *v)* self-adaptiveness.

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 [18]. The size of a zone adapts to face failures. Replicas responsible of zones of the same row or column form a quorum—similar quorums have been proved optimal [6, 15]. The overlay size is far lower than the system size and communication is restricted to replicas responsible of two abutting sub-zones. This makes our approach scalable. The overlay reacts to the varying load implied by numerous clients first by balancing it through unloaded

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC '07, March 11-15, 2007 Seoul, Korea
Copyright 2007 ACM 1-59593-480-4 /07/0003 ...$5.00.

replicas and then by expanding it. The full version of this paper is available as a technical report [4].

Background. Quorums for dynamic systems appeared in [10, 13, 2, 17, 16, 9, 8, 7]. Herlihy [10] proposes quorum modification to cope with failures. In [13, 9, 8] quorum systems are subsequently replaced by independent ones to cope with permanent failures. In [2, 17, 16, 19], a quorum relies on a specific dynamic structure, and quorum probes are said *adaptive* (they contact the quorum members successively in a reactive manner). [2] proposes quorums that intersect with high probability using a dynamic De Bruijn communication graph, [17] proposes a planar overlay where a node communicates with its closest neighbor in the plane to probe a quorum, and [16] proposes a dynamic tree-structure where And/Or primitives are used to determine quorum participants when descending into the tree. In [19], 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.

Robust emulation of shared memory in message-passing systems appeared in [5]. Since then, [3, 13, 9, 8] have implemented atomicity in dynamic systems. First, the architecture of [3] provides an atomic service that locks operations when a value is being written to prevent from reading stale values. Consequently, high concurrency might delay such operations for an arbitrarily long period of time. The work presented in [13, 9, 8] provide quorum reconfiguration mechanisms to cope with permanent changes. These papers have a common design seed: they propose quorum system replacement, thus replacing the whole quorums participants by others. Unfortunately, this global change requires message exchanges among all participants of previous and new configurations to ensure consistency. A work where dynamic quorums are maintained without global communication is [7]. This work describes an architecture for quorum systems that deal with dynamicity by exploting the fault-tolerance and self-organization capabilities of an underlying structured overlay network infrastructure. Differently from our contribution, this work focuses on defining different quorum systems deployed over the general architecture and comparing their performance. Quorums are not self-adaptive with respect to varying request load.

The paper is organized as follows. Section 2 presents the system model and introduces some preliminary notations. The description of the *Square* algorithm appears in Section 3. Section 4 shows the properties of *Square* by extensive simulations. Section 5 concludes.

2. SYSTEM MODEL AND NOTATIONS

The system contains a set of uniquely identified nodes. The set of all identifiers is denoted I. We consider a subset of nodes, called *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. Clients can access these objects infinitely often, and concurrently. However during a finite period of time, the level of concurrency is finite. Nodes do not necessarily know other nodes. Nodes can join or crash at any time and a joining node is always considered as a new node.

Atomic Consistency. 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 [14]. 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 [11]. 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 correctness of the whole application. By the locality property, we limit the description of *Square* to a single object, implementation of multiple objects being identical.

Self-Healing and Self-Adaptiveness. Two fundamental challenges arise from scalability and dynamism: (i) guaranteeing object persistence and availability, and (ii) supporting unpredictable load variations. This paper proposes a solution dedicated to both problems by satisfying the following two properties: *Self-healing* which 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 nodes (i.e., *replicas*), and by replacing failed or left ones by new ones. *Self-adaptiveness* which 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. $\mathcal{L}_i(t) \leq \mathcal{B}^i_{min}$). Self-adjustment guarantees that at any time, the load at any replica $\mathcal{L}_i(t)$ is such that $\mathcal{B}^i_{min} < \mathcal{L}_i(t) < \mathcal{B}^i_{max}$. It is noteworthy that our algorithm does not depend on this specific definition of load. Any definition of load can be used without altering the algorithm behaviour.

Dynamic Quorums. 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 intersects. We consider 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.

Replicas share a same logical overlay organized in a torus topology (as for example CAN [18]). 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 adjacent zones. These zones are 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 [17, 18] 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, 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. We refer to a replica zone (or simply to a replica) r as the product of two intervals: $I_x^r = [r.xmin, r.xmax)$ and $I_y^r = [r.ymin, r.ymax)$, where $r.xmin$ (resp. $r.xmax$) is the left-most (resp. right-most) abscissa of zone r, and $r.ymin$ (resp. $r.ymax$) is the bottom-most (resp. top-most) abscissa of zone r.

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

DEFINITION 2.1 (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 $\{r \in I \mid r.xmax > c \geq r.xmin\}$.*

THEOREM 2.2. *For any horizontal quorum $Q_{h,c}$ and any vertical quorum $Q_{v,c'}$, the intersection 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. 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.

3. ALGORITHM DESCRIPTION

Read/Write Operations. 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 of 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, $\mathcal{R}.type$ is set to write and $\mathcal{R}.value$ 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:      ⟨timestamp, value⟩ ← 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 (as described later). 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. Briefly, the traversal consists in traversing the overlay vertically or horizontally in order to contact a quorum of replicas. It ensures that a quorum is aware of the pending operation, and of the current object value.

More precisely, the Traversal protocol consists in two procedures, called respectively **Consult** and **Propagate**; the former consults a whole horizontal quorum to learn about the most up-to-date value of the object and an associated timestamp whereas the latter one propagates a value (either the one initialized by the client or the one previously consulted) and the updated timestamp to a whole vertical quorum. Each of these procedures is executed from neighbor to neighbor by forwarding the information about the request $\mathcal{R}$, until the horizontal and vertical quorums 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, knowing that a whole quorum has participated. From this point on, i can continue the operation execution, by starting a **Propagate** phase if needed (see below the fast adaptive read operations) otherwise by directly sending the response to the requesting client if operation $\mathcal{R}$ is complete. Both procedures are executed only if i is *available*, i.e., is not already involved in a dynamic event (see below).

There are two differences between **Consult** and **Propagate**. First, **Consult** gathers the most up-to-date value-timestamp pair of all the horizontal quorum replicas whereas **Propagate** updates the value-timestamp pair at all replicas of the vertical quorum. Second, **Consult** contacts each member of the quorum once following a single direction, while **Propagate** contacts each member of the quorum twice with messages sent in both directions. 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 vertical quorum. This permits later read operations to complete after a single phase without propagating this value once again.

Indeed, not only the traversal is lock-free compared to [3], but it does not require the confirmation phase of [9, 8], while proposing fast *adaptive read operations*: such operations are called fast since they require a single phase instead of two. 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 [12] 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 vertical quorum, a **Propagate** phase is required after the end of the **Consult** phase and before the read can return v. Conversely, if value v has been propagated at a whole vertical quorum, 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.

The **Thwart** protocol is executed if i receives an operation re-

quest while it is overloaded. This mechanism checks the load of each quorum until it finds a non-overloaded one. For this purpose a sequence of quorum representatives, located on the same diagonal axis, are contacted in turn. Note 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, 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 as long as its initiator is not overloaded.

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 replicas to satisfy these requests. Conversely, if some replicas of the overlay are rarely requested, then the overlay **Shrink**s to speed up operation executions. Finally, when some replicas leave the system or crash, then a **FailureDetection** requires some of the replicas around the failure to reconfigure. 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 exploits a heartbeat protocol to monitor replica vivacity. Based on this protocol, failures are detected 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 [18]. 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 $available$.

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. If this occurs, i locally decides to give away its responsibility and leave the overlay. 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, then i decides to expand the overlay. From this point on, initiator i becomes $unavailable$ (preventing itself from participating in traversals): it chooses a node j outside the memory and actively replicates its timestamp and value at j. That is, j becomes a replica, i shares a part of its own workload and responsibility zone, and j and i notify their neighbors becoming newly $available$.

4. 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. Peersim is a simulator especially suited for self-organizing large-scale systems. We used its event-based simulation mode in order to simulate asynchronous communication and independent nodes activities.

4.1 Environment

We simulate a p2p 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**).
- Once for every 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, except otherwise mentioned, requests are issued at some rate during a fixed period, after which the requests stop. To absorb the load induced by the requests, the overlay replicates the object in nodes of the system that are not yet in the memory. This self-adaptiveness occurs until the memory reaches an acceptable configuration satisfying the tradeoff between capacity and latency. An *acceptable configuration* is a 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.

Self-adaptiveness. Figure 1(a) reports the number of nodes in the memory versus time. In particular, the solid line indicates the evolution of the memory size along time, showing the adaptiveness of Square to a constant requests rate. In this figure, the memory reaches the acceptable configuration at time 9350, while the memory leaves the acceptable configuration at time $49,200$. Let us focus on the three resulting intervals. 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 joining it as replicas (**Expand**). This shows how *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, thus preserving atomicity. Figure 1(b) shows the adaptiveness of the memory to abrupt changes in load. The vertical intervals indicate the error margin at some points of the curve. We simulate a burst of load at time 23000 where the request rate is multiplied by 2. Then requests are stopped at time 46000. We clearly see that the memory is reactive and quickly self-adapts to face load variation: the memory size grows right after the burst (i.e. it is multiplied by 1.4) and shrinks right after request stops (i.e. divided by 1.2), while

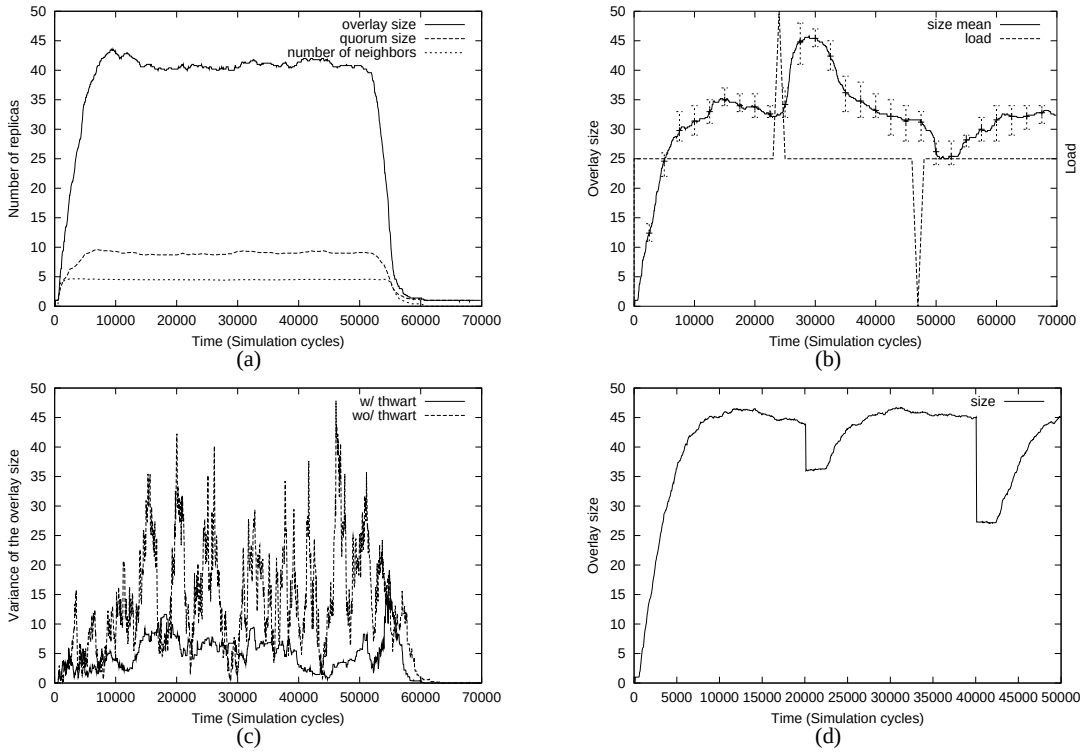


Figure 1: (a) Memory size, quorum size, and number of neighbors. (b) Self-adaptiveness in face of bursts of load. (c) Thwart impact. (d) Self-adaptiveness in face of important failures.

recovering a steady progress.

Scalability. The dotted line in Figure 1(a) 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 a skewed distribution. Since only a local neighborhood of limited-size has to be maintained, the reconfiguration needed to face dynamism is scalable.

Load-balancing. The main objective of the thwart mechanism is to balance the load among nodes. 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 1(c). 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 reduced 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. This phenomenon produces a strong variation in the memory size: many underloaded replicas of the memory shrink while many overloaded replicas expand. Conversely, with the thwart mechanism any replica tries to balance the load over the whole memory, verifying that the memory is globally overloaded before triggering an expansion. This makes the memory more stable.

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 1(d) 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 again, trying to newly face the constant request rate. This phenomenon is even more pronounced 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 have occurred.

Operation latency. Experiment of Figure 2 is composed of 5 simulations with different request rates and indicates how *Square* minimizes read operation latency. First, recall that the fast adaptive read operation contains only a **Consult** phase, thus the horizontal quorum size impacts more on read operation latency than vertical quorum size does. We tuned *Square* such that a replica that receives more read requests than write requests tends to split horizontally

request rate	read latency (in avg)	write latency (in avg)	max. memory size	max. horizontal quorum size	max. vertical quorum 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 2: Trade-off between response time and memory size.

its responsibility zone, when an expansion occurs. Since an operation is of type read with probability 0.9, replicas choose more frequently (in average) to split horizontally than vertically, consequently horizontal quorums are smaller than vertical quorums, as depicted in the 5^{th} and 6^{th} columns of Figure 2. An increase in the requests rate—indicated in column 1—strengthens this difference: it enlarges the amount of operations, thus the phenomenon becomes more evident. Furthermore, the 2^{nd} and 3^{rd} columns confirm our thought: 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.

5. CONCLUSION

This paper has proposed an atomic memory for large-scale dynamic systems. To achieve this goal the proposed algorithm presents dedicated properties. We showed these properties through extensive simulations. The originality of our approach is based on the self-adaptiveness of the memory to face the extreme dynamism of these systems and to ensure low response time: by spontaneously expanding its size when replicas become overloaded, *Square* supports bursts of load; while by quickly shrinking to the minimal number of replicas when load decreases, *Square* minimizes operation latency. By providing fast adaptive reads, *Square* is fully adapted to applications in which consultations are more common than modifications. Despite the complexity of these systems, this paper highlighted that atomic consistency is achievable without jeopardizing scalability, load-balancing, fault-tolerance, and self-adaptiveness.

Acknowledgment

We wish to thank Maria Gradinariu, Sylvestre Cozic, Romaric Ludinard, and the anonymous referees for their help.

6. REFERENCES

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

[2] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. *Distrib. Computing*, 18(2):113–124, 2005.

[3] 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, 2005.

[4] E. Anceaume, V. Gramoli, and A. Virgillito. SQUARE: Scalable quorum-based atomic memory with local reconfiguration. Technical Report 1805, IRISA Campus de Beaulieu, Rennes, France, 2006.

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

[6] B. Awerbuch and P. Vitanyi. Atomic shared register access by asynchronous hardware. In *Proc. of 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 233–243, 1986.

[7] R. Baldoni, R. Jimnez-Peris, M. Patio-Martinez, L. Querzoni, and A. Virgillito. Dynamic Quorums for DHT-based P2P Networks. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA05), Cambridge, MA, USA*, 2005.

[8] 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 (OPODIS)*, 2005.

[9] 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 (DISC)*, pages 306–320, 2003.

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

[11] 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.

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

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

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

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

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

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

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

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