On the Robustness of (Semi)Fast Quorum-Based Implementations of Atomic Shared Memory *

Chryssis Georgiou[1] **, Nicolas C. Nicolaou[2], and Alexander A. Shvartsman[23]

[1] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[2] Department of Computer Science and Engineering, University of Connecticut, Storrs, USA
[3] Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA

Abstract. This paper studies a trade-off between fault-tolerance and latency in implementations of atomic read/write objects in message-passing systems. In particular, considering *fast* or *semifast quorum-based* implementations, that is, implementations where *all* or respectively *most* read and write operations complete in a single communication round-trip, it is shown that such implementations are *not robust* due to the fact that they necessarily require a quorum system with a common intersection between its quorums.
To trade speed for fault-tolerance, the notion of *weak-semifast* implementations is introduced. Here more than a single complete slow (two round-trip) read operation is allowed for each write operation (semifast implementations allow only one such slow read). A quorum-based algorithm is given next and it is formally shown that it constitutes a weak-semifast implementation of atomic registers. The algorithm uses the notion of *Quorum Views* to facilitate the characterization of all possible object timestamp distributions that a read operation may witness during its first communication round-trip. Noteworthy is that the algorithm allows fast read operations even if they are concurrent with other read and write operations.
Finally, experimental results were gathered by simulating the algorithm using the NS-2 network simulator. The results show that under realistic conditions, less than 13% of read operations are slow, thus the overwhelming majority of operations take a single communication round-trip.

1 Introduction

Motivation and Prior Work. Atomic (linearizable) read/write memory is one of the fundamental abstractions in distributed computing. Fault-tolerant implementations of atomic objects in message-passing systems allow processes to share information with precise consistency guarantees in the presence of asynchrony and failures. A seminal implementation of atomic memory of Attiya *et al.* [1] gives a single-writer, multiple reader (SWMR) solution where each data object is replicated at n message-passing nodes. In that solution, memory access operations are guaranteed to terminate as long as the number of crashed nodes is less than $n/2$, i.e., the solution tolerates crashes of any minority of the nodes. The write protocol involves a single round-trip communication stage, while the read protocol involves two round-trip stages, where the second stage

* This work is supported in part by the NSF Grants 9988304, 0121277, and 0311368.
** The work of this author is supported in part by research funds at the University of Cyprus.

essentially performs the write of the value obtained in the first stage. Following this development, a folklore belief developed that in messaging-passing atomic memory implementations "atomic reads must write."

However, recent work by Dutta *et al.* [3] established that if the number of readers R is appropriately constrained with respect to the number of replicas S and the maximum number of crash-failures t in the system ($R < \frac{S}{t} - 2$), then single communication round-trip implementations of reads are possible. Such an implementation given in [3] is called *fast*. Subsequently, Georgiou *et al.* [9] relaxed the constraint in [3], and proposed *semifast* implementations with unbounded number of readers, where under realistic conditions most reads need only a single communication round-trip to complete. Their approach groups collections of readers into *virtual nodes*. Semifast behavior of their algorithm is preserved as long as the number of virtual nodes is appropriately restricted under $\frac{S}{t} - 2$.

Quorum systems are well-known mathematical tools that provide means for achieving coordination between processors in distributed systems [11, 6, 22, 21]. Given that the approach of Attiya *et al.* [1] is readily generalized from majorities to quorums (e.g., [20]), and that the algorithms in [3] and [9] rely on intersections in specific sets of responding servers, one may ask: *Can we characterize the conditions enabling fast implementations in a general quorum-based framework?*

This is what we establish in this work. We consider quorum-based implementations of atomic memory, and investigate the properties needed to achieve fast and semifast atomic memory implementations. Interestingly, when examining unconstrained — in terms of quorum construction and reader participation — quorum-based implementations, we discover that a *common* intersection among *all* quorum sets is necessary. This renders such implementations non-fault-tolerant, since the common intersection introduces a single point of failure. Then a natural question arises: *Was a common intersection implied in [3] and [9]?* The answer is "no", because (a) the constraint on the number of readers in [3] or virtual nodes in [9], and (b) the knowledge of the number of failures t, has the implication that the intersections of the replying sets of servers is guaranteed to consist of non-faulty processors. So, our new findings introduce complementary knowledge: *One cannot have fast or semifast implementations without common intersection unless one imposes additional constraints on the system*.

Based on this new understanding, we posed the question of whether one can avoid restrictions, such as the constraint on the number of readers or the common intersection among quorum sets, and still obtain practical and robust implementations. We show that this is indeed possible if some speed is traded for robustness. We introduce *weak-semifast* implementations that allow a greater proportion of slow reads, and develop a new algorithm that uses a predicate tool, called *quorum views*, also defined in this paper. We simulate our algorithm using the NS-2 simulator and we gather experimental results that demonstrate the practicality of our algorithm.

Related Work. Previous works extended the approach in [1] and used quorums to provide atomicity in the *multiple writer multiple reader* (MWMR) model [20, 4, 13, 5]. The work in [5] (similar to [1]) shows that the read operations must write to as many replicas as the maximum number of failures allowed. A dynamic atomic memory implementation using reconfigurable quorums is given in [18], where the sets of object replicas

can arbitrarily change over time as processes join and leave the system. Refinements of the dynamic algorithm further improved its performance in practical implementations [12, 8, 7]. When the set of replicas is not being reconfigured, the read and write protocols involve two communication round-trips. Retargeting this work to ad-hoc mobile networks, Dolev *et al.* [2] formulated the GeoQuorums approach where replicas are implemented by stationary *focal points* that in turn are implemented by mobile nodes; here quorums are composed of focal points. Interestingly, in this work some reads involve a single communication round-trip when it is confirmed that the corresponding write operation has completed.

A recent work by Guerraoui and Vukolić [15] presented a powerful notion of *Refined Quorum Systems* (RQS), where quorum members are classified in three categories, called *quorum classes*, according to their intersection size with other quorums; the first class contains quorums of large intersection, the second of smaller intersection, and the third class corresponds to traditional quorums. The authors specify the properties that the members of each quorum class must possess and use RQSs to develop an efficient *Byzantine-resilient* SWMR atomic object implementation and a solution to the consensus problem. In synchronous failure-free runs their implementation allows single communication round-trip (fast) operations. That was an improvement over a previous result from the same authors, [14], that provided bounds and imposed system restrictions to achieve robust *safe* and *regular* storage implementations in the presence of Byzantine failures. In that work they showed that *two* communication round-trips are necessary for each read operation in both safe and regular implementations even though more than $2t + 2b + 1$ servers are used, where t the maximum number of crash and b the maximum number of byzantine failures. Our work complements the work in [15] by specifying the exact properties that a *general* quorum system must possess in order to achieve single round-trip operations under *crash failures* and *asynchrony*. Furthermore we do not use quorum formation constraints such as the categories mentioned above, rather we deal with the usual quorum systems. In our implementations we only rely on client side prediction tools we call *Quorum Views*.

Malkhi and Reiter in [21] studied constructions that improve fault-tolerance of quorum systems for *byzantine failures*. They organized their constructions according to the properties quorum sets satisfy and the degree of fault-tolerance they achieve. Using such constructions they provided *safe* and *regular* ([16]) read/write register implementations. Regularity was also studied for the MWMR model in [23]; the authors presented regular implementations with single round trip operations. Peleg and Wool in [22], investigated different families of quorum systems and presented their performance in terms of *process load*, *quorum availability* (failure tolerance) and *message complexity* (quorum size). A quorum construction was then proposed that achieves high performance in comparison with prior constructions.

Our Contributions. In this paper we study the properties of quorum systems that enable communication-efficient quorum-based implementations of atomic read/write registers. In particular, we study the efficiency of *general* quorum constructions deployed in implementations with *unconstrained* number of readers. We say that an atomic SWMR implementation is *fast* if all the read and write operations complete in a single communication round-trip. A *semifast* implementation as defined in [9] allows one complete

slow read operation for each write, while the rest of read and write operations are required to be fast. In this paper we say that an implementation is *not robust* if it has a single point of failure. We consider implementations that use quorum systems $\mathbb{Q} = \{Q_i\}$, where for any two quorum sets in $\mathbb{Q}$ we have $Q_i \cap Q_j \neq \emptyset$. The contributions presented in this paper are as follows.

1. We show that fast quorum-based implementations that allow arbitrary number of readers must use certain quorum systems that necessarily render the implementations not robust. In particular we prove that a quorum-based fast implementation is possible *if and only if* the following property is satisfied by $\mathbb{Q}$: $\bigcap_{Q \in \mathbb{Q}} Q \neq \emptyset$.
In other words, there must be a *common intersection* among all quorum sets of $\mathbb{Q}$. Since a single failure in the common intersection disables the quorum system, we conclude that *robust* fast quorum-based implementations are *impossible*.

2. We then pose the natural question whether *semifast* ([9]) quorum-based implementations can be robust. We give a negative answer to this question as well: we show that *robust* semifast quorum-based implementations are also *impossible*. In particular we show that a certain property of the semifast definition ([9], Property 3) is violated when using quorum systems without a common intersection. This property states that only a *single complete* read operation is required to perform a second communication round-trip for every write operation. We prove that requiring a single complete slow read is impossible to satisfy using quorum-based implementations without a common intersection.

3. Consequently we seek implementations that enable fast reads, but permit multiple slow reads per write. We call such implementations *weak-semifast*. As a tool used in our development, we introduce the notion of *Quorum Views* that is used to characterize every possible timestamp distribution that a read operation may witness in a quorum set during its first communication round-trip. A quorum view may provide "sufficient" information on whether or not a write operation is complete. If so, then the read operation can be "fast." Otherwise the reader performs a second communication round-trip and the read operation is "slow." We define quorum views and we present an algorithm, called SLIQ (Semifast Like Implementation for Quorum systems), that makes use of the latter idea and we prove its correctness. The algorithm departs from the classic approach that implements all reads as slow, and also from the approach that allows fast reads after the completion of the write operation. In our algorithm fast reads are allowed even in the case of concurrent read and write operations.

4. We simulate our algorithm using the NS-2 network simulator and we observe that in common cases only less than 13% of the read operations need to perform a second communication round-trip, thus the overwhelming number of operations are fast.

Paper Organization. In Section 2 we present our model assumptions and definitions. In Section 3 we present the quorum system properties that are necessary to achieve fast and semifast quorum-based implementations and we show their non-robustness. The notion of quorum views and algorithm SLIQ are presented in Section 4 along with the algorithm's correctness. The results of our simulations are depicted in Section 5. We conclude in Section 6. For full proofs and additional discussion we refer the reader to [10].

2 Model and Definitions

We consider implementations for the single writer, multiple reader (SWMR) in the asynchronous message-passing model. Our system consists of three distinct sets of processes: a distinguished process w is the writer, the set of R readers with unique ids from the set $\mathcal{R} = \{r_1, \ldots, r_R\}$, and the set of S servers (where the object replicas are maintained) with unique ids from the set $\mathcal{S} = \{s_1, \ldots, s_S\}$.

A *quorum system* is a collection of sets of processes, known as *quorums*, such that every pair of such sets intersects. We define a quorum system $\mathbb{Q}$ over the set of servers $\mathcal{S}$ as follows: $\mathbb{Q} = \{Q_i : Q_i \subseteq \mathcal{S}\}$ such that for any two quorums $Q_i, Q_j \in \mathbb{Q}, Q_i \cap Q_j \neq \emptyset$. We assume that every process in the system is aware of $\mathbb{Q}$.

Our distributed system is modeled in terms of *I/O automata* [19, 17] where A_p represents the automaton A assigned to process p. Our system is composed of four kinds of automata: the writer $Writer_w$, servers $Server_{s_i}$, readers $Reader_{r_i}$, and $Channel_{p,q}$. Automata $Channel_{p,q}$ and $Channel_{q,p}$ are assumed to implement a reliable communication channels between processes $p \in \{w\} \cup \mathcal{R}$ and processes $q \in \mathcal{S}$. Each I/O automaton A_p consists of a set of states $states(A_p)$ that includes the initial state(s) of A_p, and a signature $sig(A_p)$ that specifies input, output, and internal actions that can be performed by A_p. For an action α, the tuple $\langle state, \alpha, state' \rangle$ represents the *transition* of A_p from state $state$ to $state'$ as the result of α. Such a tuple is also called a *step*, of A_p. An *execution fragment* φ of A_p is a finite or an infinite sequence $state_0, \alpha_1, state_1, \alpha_2, \ldots, \alpha_r, state_r, \ldots$ of alternating states and actions of A_p such that every $state_k, \alpha_{k+1}, state_{k+1}$ is a step of A_p. If an execution fragment begins with an initial state of A_p then it is called an *execution*. We say that an execution fragment φ' of A_p, *extends* a finite execution fragment φ of A_p if the first state of φ' is equal to the last state of φ. The concatenation of of φ and φ' is the result of the extension of φ by φ' where the duplicate occurrence of the last state of φ is eliminated. Such concatenation yields an execution fragment of A_p.

A process p *crashes* at any step $\langle state_k, \alpha_{k+1}, state_{k+1} \rangle$ in an execution ξ, if this is the last step of A_p in ξ. A process p is *faulty* in an execution ξ if p crashes in ξ; otherwise p is *correct*. A quorum $Q \in \mathbb{Q}$ is non-faulty if $\forall p \in Q$, p is correct; otherwise Q is faulty. We assume that any subset of readers, the writer, and all but one quorum in quorum system $\mathbb{Q}$ may be faulty at any execution.

2.1 Atomicity

Our goal is to implement a read/write atomic object in a message passing system by replicating the value of the object among the servers in the system. Each replica consists of a value v and an associated timestamp ts.

The client at process p may request a read operation ρ on the atomic register x by performing a $\mathsf{read}_{x,p}$ action if $p \in \mathcal{R}$. Similarly the client requests a write operation $\omega(*)$ by performing $\mathsf{write}(*)_{x,p}$ if process p is the writer. The step that includes the read or write action is called *invocation* step and the step that contains a $\mathsf{read\text{-}ack}(*)_{x,p}$ or a $\mathsf{write\text{-}ack}_{x,p}$ action is called a *response* step. An operation π is *incomplete* in an execution ξ, if ξ contains the invocation step of π but does not contain the associated response step for π; otherwise we say that π is *complete*. We assume that the requests of a client are *well-formed* meaning that the client does not request a $read$ or $write$ action on an object x, before receiving a read-ack or write-ack from a previously invoked

action on x. From this point onward we assume a single register object. By composing multiple single register implementations, one may obtain the complete atomic shared-memory [17].

In an execution we say that an operation (read or write) π_1 *precedes* another operation π_2, or π_2 *succeeds* π_1, if the response step for π_1 precedes the invocation step of π_2; this is denoted by $\pi_1 \rightarrow \pi_2$. Two operations are *concurrent* if neither precedes the other.

Correctness of an implementation of an atomic object is defined in terms of the *termination* and *atomicity* properties. The termination property requires that any operation invoked by a correct process eventually completes, provided that the failures obey our failure model. Atomicity is defined as follows [17]: Consider the set Π of all complete operations in any well-formed execution. Then there exists an irreflexive partial ordering $\prec$ on operations in Π, satisfying the following: (1) For any operation $\pi \in \Pi$, there are finitely many operations π' such that $\pi' \prec \pi$. (2) If operation π_1 precedes the operation π_2 in Π, then it cannot be the case that $\pi_2 \prec \pi_1$. (3) If π is a write operation and π' is any operation in Π, then either $\pi \prec \pi'$ or $\pi' \prec \pi$. (4) The value returned by a read operation is the value written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write).

2.2 Fast, Semifast and Weak-Semifast Implementations

We use the definition of *fast* implementation given by Dutta et al. [3], in particular we say that a read or write operation is ***fast*** if it completes in a single communication round-trip (or round for short). A fast implementation contains only fast operations in any execution. We define a communication round as follows:

Definition 1. *A process p performs a communication round during operation π if all of the following hold:*
(1) *p sends read or write messages for π to a subset of processes,*
(2) *any process p' that receives a message from p for operation π, replies to the message with a read or write acknowledgment respectively before receiving any other message*[4]*,*
(3) *when process p receives enough replies for π it responds to the client.*

The results of [3] show that in fast implementations the number of readers must be constrained with respect to the number of servers. To relax such constraints, [9] proposed *semifast* implementations where some read operations are allowed to perform two communication rounds. The definition of semifast implementations is given below, where $\mathfrak{R}(\rho)$ denotes the unique write operation that wrote the value returned by ρ:

Definition 2. *An implementation I of an atomic object is **semifast** if the following are satisfied:*
P1. *Every* write *operation is fast.*
P2. *Any complete* read *operation performs one or two communication rounds between the invocation and response.*
P3. *For any execution ξ of I, if ρ_1 is a two-round read operation, then any read operation ρ_2 with $\mathfrak{R}(\rho_1) = \mathfrak{R}(\rho_2)$, such that $\rho_1 \rightarrow \rho_2$ or $\rho_2 \rightarrow \rho_1$, must be fast.*

[4] Intuitively this property is used to stress the fact that processes do not need to wait for other messages before replying to p.

P4. *There exists an execution ξ of I that contains at least one write operation ω and at least one read operation ρ_1 with $\Re(\rho_1) = \omega$, such that all read operations ρ with $\Re(\rho) = \omega$ (including ρ_1) are fast.*

We define a new class of implementations that we call ***weak-semifast*** implementations. This class is defined in terms of properties **P1**, **P2**, and **P4** of the semifast implementations, and it does not include property **P3**. In other words, weak-semifast implementations allow multiple "slow" complete read operations for every write operation in contrast with property **P3** that allows a single such read operation. Thus the two classes are distinct.

Given that any subset of readers and the writer may crash, then termination is guaranteed only if no operation waits for replies from any reader or writer processes. Moreover our failure assumptions on the quorum system imply that no operation can wait for more than a single quorum to reply. Finally, as shown in [3, 9], fast and semifast implementations require that server processes cannot wait for more messages before replying to a read or write operation. Notice that since weak-semifast implementations share the same communication scheme in terms of communication rounds as the semifast implementations, they also follow the rules presented in this paragraph.

2.3 Quorum-Based Algorithms

The results in the next two sections pertain to atomic register implementations that have the following characteristics: (1) they use a quorum system to group the object replicas, (2) participants are aware of the quorum system construction and the operation protocol and (3) in every execution there is at least one non-faulty quorum.

Characteristic (3) describes the failure model of the algorithms we consider. Observe that: (i) according to the pairwise intersection property of quorums it suffices to obtain replies from a single quorum, and (ii) according to (3) only a single quorum may be alive in any execution of the algorithm, and thus waiting for more than one quorum before replying may affect operation termination. Thus we assume that any operation waits for exactly one quorum to reply.

3 Quorum Properties and Fast/Semifast Impossibility

A process p, that invokes an operation π, is said to *contact* a subset of servers $\mathcal{G} \subseteq \mathcal{S}$, denoted by $cnt(\mathcal{G})_{p,\pi}$, if for every server $s_i \in \mathcal{G}$: (a) s_i receives the messages sent by p within the operation π, (b) s_i replies to p, and (c) p receives the reply from s_i. If $cnt(\mathcal{G})_{p,\pi}$ and additionally no other server (i.e., $s_i \notin \mathcal{G}$) receives any message from p within the operation π then we say that p *strictly contacts* $\mathcal{G}$, and is denoted by $scnt(\mathcal{G})_{p,\pi}$. Let $maxTS$ denote the maximum timestamp that a read operation ρ_i witnesses after $cnt(\mathcal{G})_{*,\rho_i}$ or $scnt(\mathcal{G})_{*,\rho_i}$, for some $\mathcal{G} \subseteq \mathcal{S}$, during its first round.

Below we discuss our results regarding quorum-based fast and semifast implementations. More detailed analysis can be found in [10].

Fast Implementations. We now state the quorum property that is both necessary and sufficient to obtain fast quorum-based implementations.

Theorem 1. *A fast quorum-based implementation I of a read/write atomic register is possible iff the underlying quorum system $\mathbb{Q}$ satisfies:* $\bigcap_{Q \in \mathbb{Q}} Q \neq \emptyset$.

Proof (Sketch). We prove the two directions of the theorem separately. We first show that if we want fast implementations it is necessary to have common intersection, and then we show that having a common intersection it is sufficient to build fast implementations.

The first part of the proof relies on an execution construction. The construction involves an execution ξ_0 that contains a complete write operation which $scnt(Q_i)_{*,\omega}$ and a series of executions $\xi_1, \ldots, \xi_{n-2}$ ($n = |\mathbb{Q}|$). Starting from ξ_0 we extend each execution ξ_i with $i+2$ read operations such that each of them strictly contacts a different quorum. Using an induction on the number of read operations, it can be shown that atomicity is preserved only if the last read operation, say ρ_k in the execution ξ_{k-2}, may witness the maximum timestamp. Assuming that ρ_k $scnt(Q_z)_{*,\rho_k}$ and the maximum timestamp is introduced only in a quorum intersection $\mathcal{Q}_{inter}$ then ρ_k may witness the maximum timestamp only if $\mathcal{Q}_{inter} \cap Q_z \neq \emptyset$. Generalizing to the full quorum system the common intersection follows.

The fact that the common intersection is sufficient for fast implementations follows from a trivial implementation: each read/write operation contacts (only) the servers in the common intersection and returns the maximum timestamp observed in the first communication round. Notice here that according to our failure model, all the servers of the common intersection must remain alive during the execution. Thus atomicity is not violated since every read/write operation will gather all the servers in the common intersection and furthermore all operations complete in a single communication round.

Theorem 1 leads to the following result.

Theorem 2. *Fast quorum-based implementations are not robust.*

Proof. Theorem 1 requires a common intersection between the quorum sets of the quorum system $\mathbb{Q}$. If any member node s_i of the common intersection fails, then all quorum members of the quorum system are faulty since $\forall Q \in \mathbb{Q}, s_i \in Q$. Hence it follows that the quorum system $\mathbb{Q}$ fails. Therefore the quorum system suffers from a single point of failure and as a sequel it is not robust. As a result, any implementation that relies on such a quorum system is also not robust.

Semifast implementations. Since fast implementations are not possible if common intersection property is not satisfied by the quorum system, a natural question arise whether robust *semifast* implementations can be achieved. We show that robust semifast implementations are also impossible if the common intersection between the quorums is not preserved. We use the properties of the semifast implementations as presented in Definition 2.

We first prove a lemma that specifies when a read operation is necessary to perform a second communication round. The lemma is general and algorithm-independent.

Lemma 1. *A read operation ρ from a reader r that $scnt(Q_i)_{r,\rho}, Q_i \in \mathbb{Q}$ cannot be fast if $\exists s \in Q_i : s.ts < maxTS$ and $\exists \mathcal{Q} \subset \mathbb{Q}$ s.t. $Q_i \cap \left(\bigcap_{q \in \mathcal{Q}} q\right) \neq \emptyset$ and $\forall s \in Q_i \cap \left(\bigcap_{q \in \mathcal{Q}} q\right), s.ts = maxTS$.*

We can then derive the following result.

Theorem 3. *No quorum-based semifast implementation is possible if $\bigcap_{Q \in \mathbb{Q}} Q = \emptyset$.*

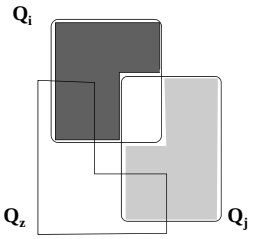


Fig. 1. Intersections of three quorums Q_i, Q_j, Q_z.

Proof (Sketch). The proof is build upon execution constructions that exploit a basic quorum system similar to the one presented in Figure 1 with no common intersection. Based on that figure, consider an execution that contains a write operation ω and three read operations ρ_1, ρ_2 and ρ_3. Assume that the write is incomplete and $scnt(Q_j \cap Q_i)_{w,\omega}$. Moreover the ρ_1 operation $scnt(Q_i)_{*,\rho_1}$ during its first communication round. According to Lemma 1, ρ_1 needs to perform a second communication round and suppose it $scnt(Q_i \cap Q_j)_{*,\rho_1}$ before the first communication round of ρ_2. Thus when ρ_2 is executed, and $scnt(Q_j)_{*,\rho_2}$, observes that ρ_1 performed a second communication round but it cannot distinguish whether ρ_1 is completed or not. Here is where the key idea of the proof lies: if ρ_2 is fast (s.t. Property 3 of the semifast definition holds), then if ρ_3 $scnt(Q_z)_{*,\rho_3}$, ρ_3 will not witness the maximum timestamp and thus will return an older value violating atomicity. So to preserve atomicity ρ_2 has to proceed to a second communication round. Since, however, ρ_2 cannot distinguish between a complete second communication round of ρ_1 that $scnt(Q_i)_{*,\rho_1}$ and the incomplete second communication round of ρ_1 that $scnt(Q_i \cap Q_j)_{*,\rho_1}$, then ρ_1 and ρ_2 may not be concurrent but yet be both slow. That however violates Property 3 of the semifast definition.

We conclude that robust quorum-based semifast implementations are not possible.

Corollary 1. *Semifast quorum-based implementations are not robust.*

Remark 1. Observe that the robustness of fast quorum-based implementations can be improved by the following techniques: (i) relaxing the failure model and requiring more than a single quorum to reply at any read/write operation, and (ii) impose restrictions on the number of reader participants and on the construction of the quorum system they deploy. This however will negatively affect the performance of the quorum system and will introduce strong assumptions for its maintenance, making eventually the use of quorums impractical. Thus in this work we avoid making such assumptions and we prefer to trade operation performance for higher fault-tolerance and applicability.

The following example help us visualize the application of the second technique presented in the above remark. Assume the following setting under [3]: $S = \{1, 2, 3, 4, 5\}, R = 2$ and $t = 1$. Any operation may receive replies from $S - t$ servers and from one of the sets: $Q_1 = \{1, 2, 3, 4\}, Q_2 = \{1, 3, 4, 5\}, Q_3 = \{1, 2, 4, 5\}, Q_4 = \{1, 2, 3, 5\}, Q_5 = \{2, 3, 4, 5\}$. Observe that the intersection of any three sets contains two servers $(t + 1)$. Since there are two readers and one writer then a written value may

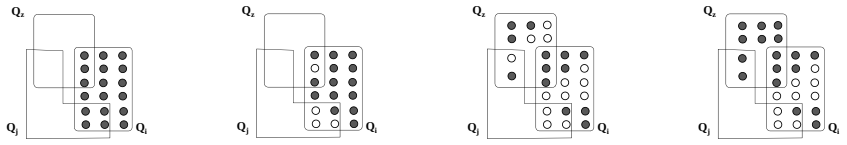


Fig. 2. (a) $qView(1)$, (b) $qView(2)$, (c) $qView(3)$ with incomplete write, (d) $qView(3)$ with complete write.

be disseminated by contacting at most three different sets in the worst case (a different set per read/write operation). So restricting the number of readers allows the concentration of the common intersection between a subset of quorum sets, which serves as a "hot spot" to ensure consistency between the operations.

4 Weak-Semifast Implementations

In the previous section we have established that no robust fast or semifast quorum-based implementations are possible. We therefore now consider weak-semifast implementations. In this section we introduce the notion of *Quorum Views* that describe certain knowledge that a read operation may witness during its first communication round. We then present an algorithm, called SLIQ, for atomic registers, and we reason, based on the knowledge in quorum views, about read operations needing to perform one or two communication rounds to complete. We deviate from the restrictive quorums properties presented in Section 3 and we allow our implementation to use an *arbitrary* quorum system construction.

4.1 Quorum Views

A *quorum view* refers to the distribution of the maximum timestamp that a read operation ρ_i witnesses after its first communication round. Consider that the read operation ρ_i strictly contacts quorum Q_i during its first communication round, denoted by $scnt(Q_i)_{*,\rho_i}$. Each member $s \in Q_i$ replies with a timestamp $s.ts$ to ρ_i. We define quorum views in terms of the following three possible cases for ρ_i:

1. $[qView(1)]$ $\forall s \in Q_i : s.ts = maxTS$,
2. $[qView(2)]$ $\forall Q_j \in \mathbb{Q}, i \neq j, \exists A \subseteq Q_i \cap Q_j$, s.t. $A \neq \emptyset$ and $\forall s \in A : s.ts < maxTS$,
3. $[qView(3)]$ $\exists s' \in Q_i : s'.ts < maxTS$ and $\exists Q_j \in \mathbb{Q}, i \neq j$ s.t. $\forall s \in Q_i \cap Q_j : s.ts = maxTS$.

Analyzing these three types of quorum views we can derive conclusions on the state of the write operation (complete or incomplete) that tries to propagate a value with the $maxTS$ in the system. Figure 2 illustrates those quorum views assuming that the read operation ρ, $scnt(Q_i)_{*,\rho}$. The dark nodes maintain the maximum timestamp of the system and white nodes or "empty" quorums maintain an older timestamp. Recall that it follows from our failure model that no operation (read or write) can wait for more than one quorum to reply. Thus having a full quorum reporting the same $maxTS$, as

seen in Fig. 2(a), implies the possible completion of the write operation (in the case of Figure 2(a) the complete write operation strictly contacts Q_i).

Observe that if a full quorum contains $maxTS$ then the members of any intersection of that quorum contain $maxTS$. So witnessing a subset of members of each intersection of Q_i (as seen in Fig. 2(b) the representation of $qView(2)$) to maintain an older timestamp, implies directly that the write operation which propagates $maxTS$ is not yet complete.

Finally, $qView(3)$, provides insufficient information regarding the state of the write operation. Observe Figures 2(c) and 2(d). In the former an incomplete write operation propagates the $maxTS$ in the dark nodes and in the latter it completes by receiving replies from Q_z. Notice that if a read operation ρ strictly contacts Q_i (i.e., $scnt(Q_i)_{*,\rho}$) in the two executions, it won't be able to distinguish 2(c) from 2(d). So, more formally, if an operation witnesses some intersection $Q_i \cap Q_z$ that contains $maxTS$ in all of its members, then a write operation might: (i) have been completed and contacted Q_z or (ii) be incomplete and contacted a subset of servers B such that $Q_i \cap Q_z \subseteq B$ and $\forall Q_j \in \mathbb{Q}, Q_j \not\subseteq B$.

4.2 Algorithm SLIQ

Our implementation includes, automaton $Writer_w$ that handles the write operations for the writer process w, automaton $Reader_{r_i}$ that handles the reading for each $r_i \in \mathcal{R}$, and automaton $Server_{s_i}$ that handles the read and write requests on the atomic register for each $s_i \in \mathcal{S}$. These automata use reliable asynchronous process-to-process channels $Channel_{p,q}$ to communicate.

Algorithm Description. Due to space limitations we only provide a high level description of our algorithm. A more technical description and formal specification can be found in [10].

Writer. The write protocol involves the propagation of a write message to all the servers. Once the writer receives replies from a full quorum it increments its timestamp and the operation completes.

Readers. The read protocol also requires that a reader propagates a read message to all the servers. Once the reader receives replies from a full quorum it examines the maximum timestamp ($maxTS$) distribution within that quorum, which in turn characterizes a quorum view. If the view is either $qView(1)$ or $qView(2)$ then the reader terminates in the first communication round and returns $maxTS$ or $maxTS - 1$ respectively. If the view is $qView(3)$ then the reader proceeds to the second communication round where it propagates the maximum timestamp to a full quorum in a similar manner as the writer. Once the reader gets replies from a full quorum, the operation completes, returning $maxTS$.

Servers. The servers maintain a passive role; they just receive messages, update their replica value according to the message contents and reply to those messages.

Algorithm Correctness. We prove that algorithm SLIQ follows its specifications and preserves atomicity; specifically we show that each atomicity property of Section 2.1 holds for any execution of the algorithm. The main theorem is the following:

Theorem 4. *Algorithm* SLIQ *implements a SWMR atomic read/write register.*

Proof (Sketch). Let $ts_p(\pi)$ to denote the timestamp of the read or write operation π from a process p, after the completion event of π. Summarizing the atomicity properties we want to show: (i) The timestamp at each process is monotonically increasing, (ii) If a read operation ρ succeeds a write operation ω then $ts_*(\rho) \geq ts_w(\omega)$ and, (iii) If ρ_1 and ρ_2 are two read operations such that $\rho_1 \rightarrow \rho_2$ then $ts_*(\rho_2) \geq ts_*(\rho_1)$. The monotonicity on the timestamps (property (i)) for every process can be easily derived from the algorithm. For property (ii) we can observe that since the write operation is completed then ρ will witness a timestamp ts greater or equal to $ts_w(\omega)$. If $ts > ts_w(\omega)$ then $ts_*(\rho) \geq ts_w(\omega)$ since $ts_*(\rho) = ts$ or $ts_*(\rho) = ts - 1$. If $ts = ts_w(\omega)$ then ρ will witness either $qView(1)$ or $qView(3)$. In either case $ts_*(\rho) = ts$. Finally for property (iii) we investigate all the possible quorum views. We need to show that if ρ_1 is fast then $ts_*(\rho_2)$ is at least equal to $ts_*(\rho_1)$. Notice that ρ_1 is fast only if a $qView(1)$ or $qView(2)$ is observed. If $qView(1)$ is witnessed by ρ_1 then ρ_2 witnesses an intersection with timestamps greater or equal to $ts_*(\rho_1)$. If $qView(2)$ is witnessed by ρ_1 then a timestamp $ts = ts_*(\rho_1) + 1$ has already introduced in the system and thus the write operation that wrote a timestamp equal to $ts_*(\rho_1)$ is already completed. Thus, by Property (ii) ρ_2 returns $ts_*(\rho_2) \geq ts_*(\rho_1)$, and hence property (iii) follows.

Note that it is straightforward to verify that SLIQ belongs in the class of weak-semifast implementations (that is, it satisfies properties **P1, P2** and **P4** of Definition 2).

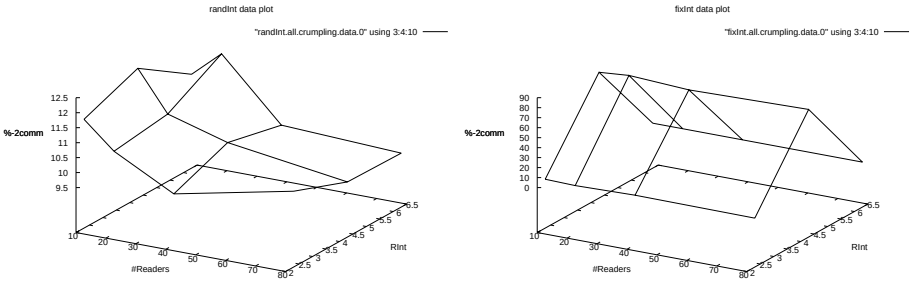
5 Simulation Results

To practically evaluate our findings, we simulated our algorithm using the the NS-2 network simulator. The detailed testbed and discussion regarding the simulation appears in [10]. According to our setting, only the messages between the invoking processes and the servers, and the replies from the servers are delivered (no messages are exchanged between any servers or among the invoking processes).

We have evaluated our approach over multiple quorum systems (majorities $\mathbb{Q}_m$, matrix quorums $\mathbb{Q}_x$ and crumbling walls $\mathbb{Q}_c$), but due to space limitations we only present here some of the plots we obtained exploiting crumbling walls (see [22]). The quorum system is generated apriori and is distributed to each participant node via an external service (out of the scope of this work). No dynamic quorums are assumed, so the configuration of the quorum system remains the same throughout the execution of the simulation. We model server failures by choosing the non-faulty quorum and allowing any server that is not a member of that quorum to fail by crashing. Note that the non-faulty quorum is not known to any of the participants. The positive time parameter $cInt$ is used, to model the failure frequency or reliability of every server s_i.

We use the positive time parameters $rInt$ and $wInt$ (both greater than 1 sec) to model the time intervals between any two successive read operations and any two successive write operations respectively. We considered three simulation scenarios corresponding to the following parameters: (i) $rInt < wInt$: this models frequent reads and infrequent writes, (ii) $rInt = wInt$: this models evenly spaced reads and writes, (iii) $rInt > wInt$: this models infrequent reads and frequent writes.

Furthermore for each one of the above scenarios we consider two settings:



Setting a: Stochastic simulations Setting b: Fixed interval simulations

Fig. 3. Simple runs using Crumbling Walls

(a) *Stochastic setting*: the read/write intervals vary randomly within $[0 \dots rInt]$ and $[0 \dots wInt]$ respectively.

(b) *Fixed setting*: the read/write intervals are fixed to the value of $rInt$ and $wInt$ respectively.

We can summarize our simulations testbed for each class of quorums and for the settings presented above, as follows:

(1) **Simple Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x,\ \mathbb{Q}_m$)** $|\mathcal{S}| = 25$ ($\mathbb{Q}_c,\ \mathbb{Q}_x$) or $|\mathcal{S}| = 10$ ($\mathbb{Q}_m$), $cInt = 0$ (failure check for every reply) and $|\mathcal{R}| \in [10, 20, 40, 80]$. Here we want to demonstrate the performance of the algorithm under similar environments (quorum,failures) but with different read load.

(2) **Quorum Diversity Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x$)** $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_c$) and $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_x$), $cInt = 0$ and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs demonstrate the performance of the algorithm in different quorum systems with varying quorum membership. Each quorum is tested in variable read load.

(3) **Failure Diversity Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x$)** $|\mathcal{S}| = 25$, $cInt \in [10 \dots 50]$ with steps of 10 and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs tested the durability of the algorithm to failures. Notice that the smaller the crash interval the faster we diverge to the non-faulty quorum. As the crash interval becomes bigger, less servers fail and thus more quorums "survive" in the quorum system. For this class of runs we tested both the cases when the servers get the crash interval randomly from $[0 \dots cInt]$ and $[10 \dots 10 + cInt]$.

Figure 3 illustrates the results obtained when we assumed simple runs and exploiting crumbling walls quorum. The Z axis presents the percentage of the read operations that performed two communication rounds, the X axis corresponds to the number of reader participants and the Y axis represents time and in particular the $rInt$ interval. In the stochastic environment (Figure 3.a) we observe that the percentage of slow reads drops as the number of readers increases, regardless of the value of $rInt$. This behavior can be explained from the fact that the concurrency between the operations is minimized and thus the maximum timestamp is propagated (by both the writer and the readers) to enough servers that favor the fast behavior. Since the convergence point is similar

regardless the number of readers, then increasing the readers, increases the number of fast reads and decreases the percentage of slow reads. Similar behavior is observed in the fixed interval environment (Figure 3.b) whenever there is no strict concurrency between the reads and the writes. The worst case is observed at the point where all operations are invoked concurrently.

Our results (including the ones given in [10]) reveal that in realistic cases (i.e. stochastic settings), the percentage of two communication round reads does not exceed 13%. The only case that requires more than 85% of the reads to be slow is the worst case scenario were the read and write intervals are fixed to the same value. Notice however that this scenario is unlikely to appear in practical settings. Comparing our results with the ones obtained in [9] one can observe that the difference in the random scenarios does not exceed 6%.

6 Conclusions

In this paper we have shown that no robust fast or semifast quorum-based implementations of atomic read/write objects are possible in the presence of crashes. We thus introduced the notion of weak-semifast implementations, reasoned that this notion is meaningful, and showed that robust weak-semifast quorum-based implementations exist. As a tool, we introduced the notion of a Quorum View that we used in the design and analysis of our robust algorithm. We formally proved the correctness of the algorithm and we obtained simulation results that demonstrate that under realistic conditions the overwhelming number of read operations are fast.

The algorithm does not explicitly provide any guarantees on the relative frequency of slow and fast read operations. Thus it would be interesting to examine ways to reduce the number of slow operations either by imposing a supplemental communication scheme or by using a special form of quorum systems. An interesting direction is to investigate whether combining quorum views with refined quorum systems [15] can lead to more efficient implementations. In another direction, dynamic membership of quorum systems can also further improve the flexibility and fault tolerance of quorum-based implementations. Given the results in [4, 18] a quorum reconfiguration requires some communication overhead. So a natural question arises regarding the communication efficiency of such dynamic systems and their impact on the performance of a weak-semifast implementation.

References

1. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.
2. S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 306–320, 2003.
3. P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 236–245, 2004.
4. B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 454–463, 2000.

5. R. Fan and N. Lynch. Efficient replication of large data objects. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 75–91, 2003.
6. H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.
7. C. Georgiou, P. M. Musial, and A. A. Shvartsman. Developing a consistent domain-oriented distributed object service. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 149–158, 2005.
8. C. Georgiou, P. M. Musial, and A. A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science*, 383(1):59–85, 2007.
9. C. Georgiou, N. Nicolaou, and A. Shvartsman. Fault-tolerant semifast implementations for atomic read/write registers. *Journal of Parallel and Distributed Computing*, accepted, 2008. (A preliminary version appears in SPAA 2006, pages 281–290.)
10. C. Georgiou, N. Nicolaou, and A. Shvartsman. On the robustness of (semi)fast quorum-based implementations of atomic shared memory, 2008. `http://www.cse.uconn.edu/~ncn03001/pubs/TRs/GNS08.pdf`.
11. D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.
12. S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, pages 259–268, 2003.
13. V. Gramoli, E. Anceaume, and A. Virgillito. SQUARE: scalable quorum-based atomic memory with local reconfiguration. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 574–579, 2007.
14. R. Guerraoui and M. Vukolić. How fast can a very robust read be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 248–257, 2006.
15. R. Guerraoui and M. Vukolić. Refined quorum systems. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2007.
16. L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.
17. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
18. N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, 2002.
19. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.
20. N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, 1997.
21. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
22. D. Peleg and A. Wool. Crumbling walls: A class of high availability quorum systems. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 120–129, 1995.
23. C. Shao, E. Pierce, and J. L. Welch. Multi-writer consistency conditions for shared memory objects. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 106–120, 2003.