On the Efficiency of Atomic Multi-Reader, Multi-Writer Distributed Memory

Burkhard Englert[1], Chryssis Georgiou[2], Peter M. Musial[3] *, Nicolas Nicolaou[4], and Alexander A. Shvartsman[4] **

[1] Comp. Engineering and Comp. Science, California State University Long Beach
[2] Department of Computer Science, University of Cyprus
[3] Department of Computer Science, University of Puerto Rico
[4] Computer Science and Engineering, University of Connecticut

Abstract. This paper considers quorum-replicated, multi-writer, multi-reader (MWMR) implementations of survivable atomic registers in a distributed message-passing system with processors prone to failures. Previous implementations in such settings invariably required two rounds of communication between readers/writers and replica owners. Hence the question arises whether it is possible to have single round read and/or write operations in this setting.
We thus devise an algorithm, called SFW, that exploits a new technique called *server side ordering* (SSO), which –unlike previous approaches– places partial responsibility for the ordering of write operations on the replica owners (the servers). With SSO, *fast* write operations are introduced for the very first time in the MWMR setting. We prove that our algorithm preserves atomicity in all permissible executions. While algorithm SFW shows that in principle fast writes are possible, we also show that under certain conditions the MWMR model imposes inherent limitations on any quorum-based fast write implementation of a *safe read/write register* and potentially even restricts the number of writer participants in the system. In this case our algorithm achieves near optimal efficiency.

1 Introduction

Data survivability is essential in distributed systems. Replication is broadly used to sustain critical data in networked settings prone to failures, and a variety of distributed storage systems have been designed to replicate and maintain data residing at distinct network locations or servers. Together with replication come the problems of maintaining consistency among the replicas and of efficiency of access to data, all in the presence of failures.

A long string of research has been addressing the consistency challenge by devising efficient, wait-free, atomic (linearizable [22]) read/write sharable objects in

* Part of this work was performed while the author was affiliated with: Comp. Sci. Dept., Naval Postgraduate School, USA.

** Work supported in part by the NSF awards 0702670, 0121277, and 0311368.

message-passing systems (e.g., [1–3, 6, 8, 15, 16, 21, 24, 27]). An atomic read/write object, or register [23], provides the semantics of a sequentially accessed single object. The underlying implementations replicate the object at several failure-prone servers and allow concurrent reading and writing by failure-prone clients. The *efficiency* of read or write operations is measured in terms of the number of communication rounds between clients and servers.

Prior Work. In the SWMR model, Attiya et al. [3] achieve consistency by exploiting the intersecting sets of majorities in combination with $\langle timestamp, value \rangle$ pairs, comprised of a logical clock and the associated replica value. A write operation increments the writer's local timestamp and delivers the new timestamp-value pair to a majority of servers, taking one round. A read operation obtains timestamp-value pairs from some majority, then propagates the pair corresponding to the highest timestamp to some majority of servers, thus taking two rounds.Avoiding the second communication round may lead to violations of atomicity when reads are concurrent with a write.

The majority-based approach in [3] is readily generalized to quorum-based approaches (e.g., [27, 9, 24, 10, 19]). In this context, a *quorum system* [17, 30, 12, 29, 28] is a collection of subsets of server identifiers with pairwise non-empty intersections. The work of [10] shows that the read operations must write to as many replica servers as the maximum number of failures allowed. A dynamic atomic memory implementation using reconfigurable quorums is given in [24] (with several practical refinements in [18, 13, 14, 5]), where the sets of servers can arbitrarily change over time as processes join and leave the system. When the set of servers is not being reconfigured, the read and write protocols involve two communication rounds. Retargeting this work to ad-hoc mobile networks, Dolev *et al.* [7] formulated the GeoQuorums approach. There (and in [5]), some reads involve a single communication round when it is confirmed that the corresponding write operation has completed.

Starting from [3] a common folklore belief developed that "atomic reads must write". Dutta et al. [8] present the first *fast* atomic SWMR implementation where all operations take a *single* communication round. They show that fast behavior is achievable only when the number of reader processes R is inferior to $\frac{S}{t} - 2$, where S is the number of servers, t of whom may crash. They also showed that fast implementations in the MWMR model are impossible in the presence of a single server failure. Georgiou et al. [16] introduced the notion of *virtual nodes* that enables an unbounded number of readers. They define the notion of *semifast* implementations were only a single read operation per write needs to be "slow" (take two rounds). Their algorithm requires that the number of virtual nodes V is inferior to $\frac{S}{t} - 2$; this does not prevent multiple readers as long as at least one virtual node exists. They also show that semifast MWMR implementations are impossible.

Other works, e.g., [1, 20, 21, 15], pursue bounds on the efficiency of distributed storage in a variety of organizational and failure models. For example, [20, 1], explore conditions under which two round operations are required by safe and regular SWMR registers.

Recently quorum-based approaches were further explored in the context of efficient atomic registers [21, 15]. Guerraoui and Vukolić [21] defined the notion of *Refined Quorum Systems* (RQS), where quorums are classified in three categories, according to their intersection size with other quorums. The authors characterize these properties and develop an efficient Byzantine-resilient SWMR atomic object implementation and a solution to the consensus problem. In synchronous failure-free runs their implementation is fast. Georgiou et al. [15] specified the properties that a general quorum system must possess in order to achieve single round operations in the presence of crashes and asynchrony. They showed that fast and semifast quorum-based SWMR implementations are possible iff a common intersection exists among all quorums, hence a single point of failure exists in such solutions (i.e., any server in the common intersection), making such implementations not robust. To trade efficiency for improved fault-tolerance, *weak-semifast* implementations are introduced in [15] that require at least one single slow read per write operation, and where all writes are fast. In addition, they present a client-side prediction tool called *Quorum Views* that enables fast read operations in general quorum-based implementations even under read/write concurrency. Simulation results demonstrated the effectiveness of this approach, showing that a small fraction of read operations need to be slow under realistic scenarios. A question that naturally follows is whether it is possible to have weak-semifast atomic MWMR register implementations.

Contributions. Intrigued by the above developments this work aims to answer the following question: *Under what conditions may one obtain efficient atomic read/write register implementations in the MWMR model?* To this end, we incorporate a new technique that enables single communication round, i.e., fast, write and read operations in that model. Our contributions are as follows.

1. To enable *fast write* operations we introduce a new technique called *Server Side Ordering (SSO)* that assigns to the server processes the responsibility of maintaining and incrementing logical timestamps, that are used by both readers and writers and helps to ensure atomicity. Previous algorithms, placed this responsibility on the writer's side. (In the presence of asynchrony and failures, SSO alone does not suffice to guarantee atomicity: using SSO by itself may result in the generation of non unique timestamps for each write operation.)
2. We developed a quorum-based implementation for atomic MWMR registers, called Sfw, that (a) employs the SSO technique by having the servers assign logical timestamps to writes. and (b) ensures uniqueness of timestamps by combining them with each writer's local write ordering. This hybrid approach guarantees uniqueness of tags among the read and write participants for every written value and allows the writers and readers to reason about the state of the system. To the best of our knowledge, this is the *first* MWMR atomic register implementation that provides the possibility of fast reads *and* writes.
3. Lastly, we develop a framework for reasoning about impossibility and lower bounds for MWMR implementations. In an *n-wise* quorum system any n

quorums have a common non-empty intersection. We call two operations *consecutive* if they are complete, not concurrent, and originate at two distinct processes. Two operations are *quorum shifting* if they are consecutive and the two originating processes receive replies from two distinct quorums during these operations. We prove lower bounds on the number of consecutive, quorum shifting fast write operations that an execution of a safe register implementation may contain. We show that a safe register implementation is impossible in an n-wise quorum system, where not all quorums have a common intersection, if any execution contains more than $n-1$ consecutive, quorum shifting single round write operations. This ultimately implies that in an implementation with only fast writes there cannot be more than $n-1$ writers. Algorithm SFW is nearly optimal since it approaches this bound as it yields executions with up to $n/2$ consecutive fast write operations, while maintaining atomicity.

Document Structure. In Section 2 we present our model and definitions. The algorithm, SFW, is presented in Section 3. The inherent limitations of MWMR model and the conditions under which it is possible to obtain fast write operations, are presented in Section 4. Because of space limitations, omitted proofs can be found in the full version of the manuscript [4].

2 Model and Definitions

We consider the asynchronous message-passing model. There are three distinct finite sets of processors: a set of readers $\mathcal{R}$, a set of writers $\mathcal{W}$, and a set of $\mathcal{S}$ servers. The identifiers of all processors are unique and comparable. Communication among the processors is accomplished via reliable communication channels. Servers are arranged into intersecting sets, or *quorums*, that together form a quorum system $\mathbb{Q}$. For a set of quorums $\mathcal{A} \subseteq \mathbb{Q}$ we denote the intersection of the quorums in $\mathcal{A}$ by $I_{\mathcal{A}} = \bigcap_{Q \in \mathcal{A}} Q$. We define specializations of quorum systems based on the number of quorums that together have a non-empty intersection.

Definition 1. *A quorum system $\mathbb{Q}$ is called an **n-wise quorum system** if for any $\mathcal{A} \subseteq \mathbb{Q}$, s.t. $|\mathcal{A}| = n$ we have $I_{\mathcal{A}} \neq \emptyset$ holds. We call n the **intersection degree** of $\mathbb{Q}$.*

In a common quorum system any two quorums intersect, and so any quorum system is a *2-wise* (pairwise) quorum system. At the other extreme, a $|\mathbb{Q}|$-*wise* quorum system has a common intersection among all quorums. From the definition it follows that an *n-wise* quorum system is also a *k-wise* quorum system, for $2 \leq k \leq n$. We will organize the servers into *n-wise* quorum systems known to all the participants as needed.

Algorithms presented in this work are specified in terms of *I/O automata* [26, 25], where an algorithm A is a composition of automata A_i, each assigned to some process i. Each A_i is defined in terms of a set of states $states(A_i)$ that includes the initial state σ_0, a signature $sig(A_i)$ that specifies input, output, and internal

actions (external signature consists of only input and output actions), and *transitions*, that for each action ν gives the triple $\langle \sigma, \nu, \sigma' \rangle$ defining the transition of A_i from state σ to state σ'. Such a triple is also called a *step*. An *execution fragment* ϕ of A_i is a finite or an infinite sequence $\sigma_0, \nu_1, \sigma_1, \nu_2, \ldots, \nu_r, \sigma_r, \ldots$ of alternating states and actions, such that every $\sigma_k, \nu_{k+1}, \sigma_{k+1}$ is a step of A_i. If an execution fragment begins with an initial state of A_i then it is called an *execution*. We say that an execution fragment ϕ' of A_i, *extends* a finite execution fragment ϕ of A_i if the first state of ϕ' is the last state of ϕ. The concatenation 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, yielding an execution fragment of A_i.

A process i *crashes* in an execution ϕ if it contains a step $\langle \sigma_k, fail_i, \sigma_{k+1} \rangle$ as the last step of A_i. A process i is *faulty* in an execution ϕ if i crashes in ϕ; otherwise i is *correct*. A quorum $Q \in \mathbb{Q}$ is non-faulty if $\forall i \in Q$, i is correct; otherwise Q is faulty. We assume that at least one quorum in $\mathbb{Q}$ is non-faulty in any execution.

Atomicity. We aim to implement atomic read/write memory, where each object is replicated at servers. Each object has a unique name, x from some set X, and object values v come from some set V_x; initially each x is set to a distinguished value v_0 ($\in V_x$). Reader p requests a read operation ρ on an object x using action $\mathsf{read}_{x,p}$. Similarly a write operation is requested using action $\mathsf{write}(*)_{x,p}$ at writer p. The steps corresponding to such actions are called *invocation* steps. An operation terminates with the corresponding $\mathsf{read\text{-}ack}(*)_{x,p}$ or $\mathsf{write\text{-}ack}_{x,p}$ action; these steps are called *response* steps. An operation π is *incomplete* in an execution when the invocation step of π does not have the associated response step; otherwise we say that π is *complete*. We assume that requests made by read and write processes are *well-formed*: a process does not request a new operation until it receives the response for a previously invoked operation.

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 in real time 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 read/write object is defined in terms of the *atomicity* and *termination* properties. Assuming the failure model discussed earlier, the termination property requires that any operation invoked by a correct process eventually completes. Atomicity is defined as follows [25]. For any execution of a memory service, if all the read and the write operations that are invoked complete, then the read and write operations can be partially ordered by an ordering $\prec$, so that the following properties are satisfied:

P1. The partial order is consistent with the external order of invocation and responses, that is, there do not exist operations π_1 and π_2, such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

P2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

P3. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order, and any read operation ordered before all writes returns the initial value of the object.

For the rest of the paper we assume a single register memory system. By composing multiple single register implementations, one may obtain a complete atomic memory [25]. Thus, we omit further mention of object names.

Efficiency and Fastness. We measure the efficiency of an atomic register implementation in terms of *communication round-trips* (or simply rounds). A round is defined as follows [8, 16, 15]:

Definition 2. *Process p performs a communication round during operation π if all of the following hold:*

1. p sends request messages that are a part of π to a set of processes,

2. any process q that receives a request message from p for operation π, replies without delay.

3. when process p receives enough replies it terminates the round (either completing π or starting new round).

Operation π is *fast* [8] if it completes after its first communication round; an implementation is fast if in each execution all operations are fast. *Semifast* implementations as defined in [16] allow some read operations to perform two communication rounds. Briefly, an implementation is semifast if the following properties are satisfied: (a) writes are fast, (b) reads complete in one or two rounds, (c) only a single complete read operation is slow (two round) per write operation, and (d) there exists an execution that contains at least one write and one read operation and all operations are fast. Finally, *weak-semifast* implementations [15] satisfy properties (a), (b), and (d), but eliminate the property (c), allowing multiple slow read operations per write. As shown in [8, 16] no MWMR implementation of atomic memory can be fast or semifast. So we focus our attention on implementations where both reads and writes maybe slow. We use quorum systems and tags to maintain, and impose an ordering on, the values written to the register replicas. We say that a quorum $Q \in \mathbb{Q}$, *replies* to a process p for an operation π during a round, if $\forall s \in Q$, s receives messages during the round and replies to these messages, and p receives all of the replies.

Given that any subset of readers or writers may crash, the termination of an operation cannot depend on the progress of any other operation. Furthermore we guarantee termination only if servers' replies within a round of some operation do not depend on receipt of any message sent by other processes. Thus we can construct executions where only the messages from the invoking processes to the servers, and from the servers to the invoking processes are delivered. Lastly, to guarantee termination under the assumed failure model, no operation can wait for more than a singe quorum to reply within the processing of a single round.

3 SSO-based Algorithm

In this section we present algorithm Sfw that utilizes a technique, called SSO, to introduce fast read *and* write operations. In traditional MWMR atomic register implementations, the writer is solely responsible for incrementing the tag that imposes the ordering on the values of the register. With the new technique, and our hybrid approach, this task is now also assigned to the servers, hence the name *Server Side Ordering* (SSO).

At a first glance, SSO appears to be an intuitive and straightforward approach: servers are responsible to increment the timestamp associated with their local replica whenever they receive a write request. Yet, this technique proved to be extremely challenging. Traditionally, two phase write operations were querying the register replicas for the latest timestamp, then they were incrementing that timestamp and finally they were assigning the new timestamp to the value to be written. Such methodology established that each individual writer was responsible to decide a *single* and *unique* timestamp to be assigned to a written value. In contrary, the new technique promises to redeem the writer for the query phase by moving the association of the written value to a timestamp (and thus its ordering) to the replica owners (servers). This however introduces new complexity to the problem since *multiple* and *different* timestamps may now be assigned to the same write request (and thus the same written value). Since timestamps are used to order the write operations, then multiple timestamps for a single write imply the appearance of the same operation in different points in the execution timeline. Hence the great challenge is to provide clients with enough information so that they decide a unique ordering for each written value to avoid violation of atomicity. For this purpose we combine the server generated timestamps with writer generated operation counters.

Algorithm Sfw involves two predicates. One for the write protocol and one for the read protocol. Both protocols evaluate the distribution of a tag within the quorum that replies to a write/read operation respectively.

The algorithm, depicted in Figures 1,2 and 3, uses $\langle tag, value \rangle$ pairs to impose ordering on the values written to the register. In contrast with the traditional approach where the tag is a two field tuple, this algorithm requires the tag to be a triple. In particular the *tag* is of the form $\langle ts, wid, wc \rangle \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}$, where the fields ts and wid are used as in common tags and represent the timestamp and writer identifier respectively. Field wc represents the write operation counter and facilitates the ability to distinguish between write operations. Initially the tag is set to $\langle 0, \min(\mathcal{W}), 0 \rangle$ in every process. In contrast to ts, wc is incremented by the writer before invokes a write operation and it denotes the sequence number of that write. Recall that by our key technique, the tags (and particularly the timestamps in the tags) are incremented by the server processes. Thus if a tag was a tuple of the form $\langle ts, wid \rangle$, then two server processes s_i and s_j may associate two different tags $\langle ts_i, w \rangle$ and $\langle ts_j, w \rangle$ to a single write operation. Any operation however that witness such tags cannot distinguish whether the tags refer to a single or different write operations from w. By introducing

```
1: at each writer w
2: procedure initialization:
3: tag ← ⟨0, wid, 0⟩, wc ← 0, wid ← writer id, rCounter ← 0
4: procedure write(v)
5: wc ← wc + 1
6: send (W, tag, wc, rCounter) to all servers
7: wait until receive (WACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
8: rcvM ← {⟨s, m⟩ : m = (WACK, inprogress, confirmed, rCounter) ∧ s sent m ∧ s ∈ Q}
9: T = {⟨ts, wid, wc⟩ : ⟨ts, wid, wc⟩ ∈ m.inprogress ∧ ⟨s, m⟩ ∈ rcvM}        /* find unique tags */
10: if ∃τ, MS, A : τ ∈ T ∧ MS = {s : ⟨s, m⟩ ∈ rcvM ∧ τ ∈ m.inprogress} ∧
      A ⊆ ℚ s.t. 0 ≤ |A| ≤ n/2 − 1 ∧ I_{A∪{Q}} ⊆ MS then
11:    tag = τ
12:    if |A| =≥ max(0, n/2 − 2) then
13:       rCounter ← rCounter + 1
14:       send (RP, tag, tag.wc, rCounter) to all servers
15:       wait until receive (RPACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
16:    end if
17: else
18:    tag = max_{⟨ts,wid,wc⟩}(T); rCounter ← rCounter + 1
19:    send (RP, tag, tag.wc, rCounter) to all servers
20:    wait until receive (RPACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
21: end if
22: return(OK)
23:
```

Fig. 1. Pseudocode for Writer protocol.

```
1: at each server s
2: procedure initialization:
3: tag ← ⟨0, 0, 0⟩, inprogress ← {}, confirmed ← ⟨0, 0, 0⟩, counter[0...|ℛ| + |𝒲|] ← 0
4: procedure serve()
5: upon rcv(msgT, tag′, wc′, rCounter′) from q ∈ 𝒲 ∪ ℛ and rCounter′ ≥ counter[pid(q)] do
6: if tag < tag′ then
7:    ⟨tag.ts, tag.wid, tag.wc⟩ = ⟨tag′.ts, tag′.wid, tag′.wc⟩
8: end if
9: if msgT = W then
10:    ⟨tag.ts, tag.wid, tag.wc⟩ = ⟨tag.ts + 1, tag′.wid, wc′⟩        /* tag′.wid = pid(q) */
11:    inprogress = (inprogress − {τ : τ ∈ inprogress ∧ τ.wid = pid(q)}) ∪ ⟨tag.ts, pid(q), tag.wc⟩
12: end if
13: if confirmed < tag′ then
14:    confirmed ← tag′
15: end if
16: counter[pid(q)] ← rCounter′
17: send (ack, inprogress, confirmed, counter[pid(q)])
18:
```

Fig. 2. Pseudocode for Server protocol.

the new term the tags will become $\langle ts_i, w, wc\rangle$ and $\langle ts_j, w, wc\rangle$, and thus any operation establishes that the same write operation was assigned two different timestamps. The triples can be compared alphanumerically. In particular we say that $tag_1 > tag_2$ if $tag_1.ts > tag_2.ts$, or $tag_1.ts = tag_2.ts \wedge tag_1.wid > tag_2.wid$, or $tag_1.ts = tag_2.ts \wedge tag_1.wid = tag_2.wid \wedge tag_1.wc > tag_2.wc$.

Server. The server maintains the register replica and acts depending on the message it receives. The local state of a server process s, is defined by three local variables: (1) the tag_s variable which is the local tag stored in the server, (2) the $confirmed_s$ variable which stores the largest tag known by s that has been

```
1: at each reader r
2: procedure initialization:
3: tag ← ⟨0, 0, 0⟩, rCounter ← 0
4: procedure read()
5: rCounter ← rCounter + 1
6: send (R, tag, tag.wc, rCounter) to all servers
7: wait until receive (RACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
8: rcvM ← {⟨s, m⟩ : m = (RACK, inprogress, confirmed, rCounter) ∧ s sent m ∧ s ∈ Q}
9: maxC = max_{m∈rcvM}(m.confirmed)                    /* find the maximum confirmed tag */
10: inP = {⟨ts, wid, wc⟩ : ⟨ts, wid, wc⟩ ∈ ⋃_{⟨s,m⟩∈rcvM} m.inprogress}
11: if ∃τ, MS, B : (τ ∈ inP ∧ τ > maxC) ∧ MS = {s : ⟨s, m⟩ ∈ rcvM ∧ τ ∈ m.inprogress} ∧
    B ⊆ ℚ s.t. 0 ≤ |B| ≤ n/2 − 2 ∧ I_{B∪{Q}} ⊆ MS then
12:     tag ← τ
13:     if |B| =≥ max(0, n/2 − 2) then
14:         rCounter ← rCounter + 1
15:         send (RP, tag, tag.wc, rCounter) to all servers
16:         wait until receive (RPACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
17:     end if
18: else
19:     tag ← maxC; MC ← {s : (⟨s, m⟩ ∈ rcvM) ∧ (m.confirmed = maxC)}
20:     if ∄C : C ⊆ ℚ ∧ |C| ≤ m − 2 ∧ I_{C∪Q} ⊆ MC then
21:         rCounter ← rCounter + 1
22:         send (RP, tag, tag.wc, rCounter) to all servers
23:         wait until receive (RPACK, inprogress, confirmed, rCounter) from some quorum Q ∈ ℚ
24:     end if
25: end if
26: return(tag)
```

Fig. 3. Pseudocode for Reader protocol.

returned by some reader or writer process and, (3) the $inprogress_s$ set which includes all the latest tags assigned by s to write requests. Each server s waits to receive a read or write message from operation initiated at some process p. Where this message contains: (a) the type of the message *msgType*, (b) the last tag returned by p (*msgtag*), (c) the value to be written if p invokes a write operation or the latest value returned by p if p invokes a read operation, (d) a counter *msgwc* that specifies the sequence number of this operation if p invokes a write or is equal to *msgtag.wc* if p invokes a read, and (e) a counter to help the server distinguish between new and stale messages from p. Upon receipt of any type of message, s updates its local and confirmed tags if they are smaller than the tag enclosed in the received message. In particular if $msgtag > tag_s$ (resp. $msgtag > confirmed_s$) then s assigns $tag_s = msgtag$ (resp. $confirmed_s = msgtag$). In addition to the above updates, if s receives a WRITE message from p, then s generates a new tag $newt = \langle tag_s.ts + 1, p, msgwc \rangle$, by incrementing the timestamp included in its local timestamp by 1 and assigning the new timestamp to the write operation from p. Note that the new tag generated is greater than both tag_s and *msgtag*. The server then pairs the new tag to the value included in the write message and changes its local tag to $tag_s = newt$. Then s adds *newt* in the *inprogress* set ,and removes any tag maintained previously in that set for any write operation from p. Once it completes its local update, s acknowledges every message received by sending its $inprogress_s$ set and $confirmed_s$ variable to the requesting process.

Writer. To uniquely identify all write operations, a writer w maintains a local variable wc that is incremented each time w invokes a write operation. Essentially that variable counts the number of write operations performed by w and every such write can be identified by the tuple $\langle w, wc \rangle$, by any process in the system. To perform a write operation $\omega = \langle w, wc \rangle$, w sends messages to all of the servers and waits for a quorum of these, Q, to reply. Once enough replies arrive (each server's *inprogress* set and *confirmed* variable), w collects all of the tags assigned to ω by each server in Q. Then it applies a predicate on the collected tags. In few words the predicate is used to checks if any of the collected tags appear in some intersection of Q with at most $\frac{n}{2}-1$ (see proof sketch below why this is sufficient) other quorums, where n the intersection degree of the deployed quorum system. If there exists such a tag τ then the writer adopts τ as the tag of the value it tried to write; otherwise the writer adopts the maximum among the collected tags in the replied quorum. The writer proceeds in a second communication round to propagate the tag assigned to the written values if: (a) the predicate holds but the tag is only propagated in an intersection of Q with more than $\frac{n}{2}-2$ other quorums, or (b) the predicate does not hold. In any other case the write operation is fast and completes in a single communication round. More formally the writer predicate is the following, where $|A|$ is rounded down to the nearest integer:

> **Writer predicate for a write ω (PW):** $\exists\ \tau, A, MS$ where: $\tau \in \{\langle ., \omega \rangle : \langle ., \omega \rangle \in \mathit{inprogress}_s(\omega)\ \wedge\ s \in Q\}$, $A \subseteq \mathbb{Q}, 0 \leq |A| \leq \frac{n}{2} - 1$, and $MS = \{s : s \in Q\ \wedge\ \tau \in \mathit{inprogress}_s(\omega)\}$, s.t. either $|A| \neq 0$ and $I_A \cap Q \subseteq MS$ or $|A| = 0$ and $Q = MS$.

Reader. The main difference between reader and writer protocols is that the reader has to examine each tag assigned to *all* of the write operations contained in *inprogress* sets of the servers that replied. (In contrast, writer examines only the tags assigned only to its own write operation.)

The reader proceeds by sending messages to all the servers and waits for some quorum of these to reply. Soon as enough replies arrive, it computes the maximum confirmed tag $maxConf$, and populates the set inP with all tags from *inprgoress* set reported by each of the replying servers. Then the reader chooses the largest tag $maxT$ found in inP and checks if: (a) $maxConf \geq maxT$, or (b) whether $maxT$ satisfies a reader predicate (defined below). If neither condition is valid, then $maxT$ tag is removed from inP and $maxT$ is assigned the next largest tag in inP, then the two checks are repeated. If inP becomes empty, then $maxConf$ is returned along with its associated value. If (a) holds, then some tag that has already been returned by some process is higher than any remaining tag in inP. In this case reader returns $maxConf$ and its assigned value. If (b) holds, then reader returns the tag and the associated value that satisfies its predicate. The reader is requires to ensure that the tag is propagated in an intersection between the replied quorum and at most $\frac{n}{2}-2$ other quorums, where n the intersection degree of the quorum system. A read operation is slow and performs a second communication round if: (1) the predicate holds but the

tag is propagated in an intersection between Q and exactly $\frac{n}{2}-2$ other quorums, or (2) the reader decides to return *maxConf*, but this was received from no complete intersection or an intersection between Q and $n-1$ other quorums. The tag and the associated value that will be returned by the read operation are propagated to some quorum of servers during the second communication round. More formally the reader predicate is, where $|B|$ is rounded down to the nearest integer:

> **Reader predicate for a read ρ (PR):** $\exists\ \tau, B, MS$, where: $\max(\tau) \in \bigcup_{s \in Q_i} \mathit{inprogress}_s(\rho)$, $B \subseteq \mathbb{Q}, 0 \leq |B| \leq \frac{n}{2}-2$, and $MS = \{s : s \in Q_i \wedge \tau \in \mathit{inprogress}_s(\rho)\}$, s.t. either $|B| \neq 0$ and $I_B \cap Q_i \subseteq MS$ or $|B| = 0$ and $Q_i = MS$.

Theorem 1. *Algorithm* SFW *implements a MWMR atomic read/write register.*

Proof (Sketch). The key challenge is to show that every reader and writer process decide on a single unique tag for each write operation, despite the fact that servers may assign different tags to that same write operation. To this end, we first show that in an n-wise quorum system, if some process p obtains replies from the servers of some quorum, then p may witness only a single tag per write operation to be distributed in a k-wise intersection, for $k < \frac{n+1}{2}$.

Writer's perspective: Based on the above observation, we show that only a single (unique) tag may satisfy the write predicate (**PW**). Observe that if there is a tag τ that satisfies **PW**, then it follows that τ is distributed in an intersection of at most $\frac{n}{2}$ quorums (i.e. $\frac{n}{2}$-wise intersection, including the replying quorum); otherwise, if no tag satisfies **PW** then the write operation is associated with the unique maximum tag received by the writer.

Reader's perspective: The goal is to show that if a read ρ returns a tag τ for a write ω, then τ was also the tag assigned to ω by the writer that invoked ω. Observe that a read operation returns a tag τ for a write ω in two cases: (a) τ satisfied the reader predicate **PR**, or (b) τ was equal to the max confirmed tag. In the first case the predicate ensures that τ was distributed in an intersection of at most $\frac{n}{2}-1$ quorums (including the replying quorum). Thus the writer should have observed the tag in at least an $\frac{n}{2}$-wise intersection, and hence τ would satisfy the writer's predicate as well. Furthermore, τ should be the only tag that satisfies the two predicates since it is distributed in an intersection that consists of less than $\frac{n+1}{2}$ quorums. If the reader returns τ because of case (b) then it follows that τ was confirmed by either a reader that was about to return τ because it satisfied its predicate, or by the writer that decided to associate τ with its write operation ω. Either way this was a unique tag and thus returning the confirmed tag maintains its uniqueness.

Using the proof of uniqueness of a tag assigned to a write operation, we proceed to show that the atomic properties are satisfied. In particular we show the following: (1) the monotonicity of the tag in all participants, (2) that if a write operation proceeds a read operation then the read returns a tag greatest or equal to the one associated to the write operation, (3) If a write $\omega_1 \rightarrow \omega_2$ then ω_2 is associated with a higher tag than the tag associated to ω_1, and (4)

If there are two read operation s.t. $\rho_1 \rightarrow \rho_2$ then ρ_2 decides and returns a value associated with a higher or equal tag than the one returned by ρ_1.

4 Write Optimality

We now investigate the conditions under which it is possible for an execution of a MWMR register implementation to contain only fast write operations. In particular we show that by exploiting an *n-wise* quorum system, it is possible to have executions with only fast write operations iff a certain number of "consecutive" write operations are contained in the execution. In extend, we show that this result imposes bounds on the number of writer participants in the system. We conclude this section by exploring the relation of our findings to fast implementations like [8]. We argue that our results generalize the characteristics of such implementations. In that respect, direct application of our results on the model of [8] yield the same bounds presented in that paper. For space limitations proofs appear in the full version of the paper [4].

We consider all operations that alter the tag value at some set of servers to be write operations. In an execution, an operation π invoked by process p is said to *contact* a subset of servers $\mathcal{G} \subseteq \mathcal{S}$, denoted by $cont_p(\mathcal{G}, \pi)$, if for every server $s \in \mathcal{G}$: (a) s receives the messages sent by p within π, (b) s replies to p, and (c) p receives the reply from s. If $cont_p(\mathcal{G}, \pi)$ occurs and additionally no other server (i.e., $s \notin \mathcal{G}$) receives any message from p within π then we say that π *strictly contacts* $\mathcal{G}$, and is denoted by $scnt_p(\mathcal{G}, \pi)$. Next we give two important definitions.

Definition 3. *Two operations π_1, π_2 are **consecutive** in an execution if: (i) they are invoked from processes p_1 and p_2, s.t. $p_1 \neq p_2$, (ii) they are complete, and (iii) $\pi_1 \rightarrow \pi_2$ or $\pi_2 \rightarrow \pi_1$ (they are not concurrent).*

In lieu to the above definition, a *safe register* constitutes the weakest consistency guarantee in the chain, and is defined [23] as property **S1**: *Any read operation that is not concurrent to any write operation returns the value written by the last preceding write operation.*

Definition 4. *A set of operations Π in an execution is called **quorum shifting** if $\forall \pi_1, \pi_2 \in \Pi$ strictly contact quorums $Q', Q'' \in \mathbb{Q}$ respectively, then π_1 and π_2 are consecutive and $Q' \neq Q''$.*

Given the two definitions above, we now show the ensuing lemma.

Lemma 1. *A read operation that succeeds a set of fast write operations Π, may retrieve the latest written value only from the servers that received messages from all the write operations in Π.*

Given an *n-wise* quorum system we show that if there are $n - 1$ consecutive, quorum shifting fast write operations in an execution then safe register implementations are possible.

Lemma 2. *Any execution fragment ϕ of a safe register implementation that uses an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$, contains at most $n-1$ consecutive, quorum shifting, fast write operations for any number of writers $W \geq 2$.*

We now show that safe register implementations are not possible if we extend any execution that contains $n-1$ consecutive writes, with one more consecutive, quorum shifting write operation. It suffices to assume a very basic system consisting of two writers w_1 and w_2, and one reader r. Thus our results hold for at least two writers.

Theorem 2. *No execution fragment ϕ of a safe register implementation that exploits an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$, can contain more than $n-1$ consecutive, quorum shifting, fast write operations for any number of writers $W \geq 2$.*

Remark 1. By close investigation of the predicates of Algorithm Sfw, one can see that Sfw approaches the bound of Theorem 2, as it produces executions that contain up to $n/2$ fast consecutive write operations, while maintaining atomic consistency. Obtaining a tighter upper bound is subject of future work.

Note that Theorem 2 is not valid in the following two cases: (i) Only a single writer exists in the system, (ii) There is a common intersection among all the quorums in the quorum system. In the first case the sole writer imposes the ordering on the tags introduced in the system and in the second case that ordering is imposed by the common servers that need to be contacted by every operation. It follows by the same theorem that it is impossible to have more than $n-1$ *consecutive* fast write operations then it is also prohibited to have more than $n-1$ *concurrent* fast write operations. Since no communication between the writers is assumed and achieving agreement in an asynchronous distributed system with a single failure (on the set of concurrent writes) is impossible, by [11], then we can obtain the following corollary:

Corollary 1. *No MWMR implementation of a safe register, that exploits an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$ and contains only fast writes is possible, if $|\mathcal{W}| > n-1$.*

Moreover assuming that readers also may alter the value of the register, and thus write, then the following theorem holds:

Theorem 3. *No MWMR implementation of a safe register, that exploits an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$ and contains only fast operations is possible, if $|\mathcal{W} \cup \mathcal{R}| > n-1$.*

Recall that [8] proved the impossibility of implementations where both writes and reads are fast in the MWMR model, while Theorem 3 complements that result by presenting the exact participation conditions under which such implementations could have been possible. They also showed that in the case of a single writer (i.e. $|\mathcal{W}| = 1$), a bound $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$ is imposed on the number of readers, where f is the total number of allowed server failures. The authors assumed that $f \leq |\mathcal{S}|/2$, and they adopted the technique of communicating with $|\mathcal{S}| - f$ servers for each operation. This technique however depicts a quorum system where every member has a size of $|\mathcal{S}| - f$. The following lemma presents the intersection degree of such a system.

Lemma 3. *The intersection degree of a quorum system* $\mathbb{Q}$ *where* $\forall Q_i \in \mathbb{Q}$, $|Q_i| = |\mathcal{S}| - f$ *is equal to* $\frac{|\mathcal{S}|}{f} - 1$.

Note that by Lemma 3 and Theorem 3, the system in [8] could only accommodate:

$$|\mathcal{W} \cup \mathcal{R}| \leq (\frac{|\mathcal{S}|}{f} - 1) - 1 \Rightarrow 1 + |\mathcal{R}| \leq \frac{|\mathcal{S}|}{f} - 2 \Rightarrow |\mathcal{R}| \leq \frac{|\mathcal{S}|}{f} - 3$$

and thus their bound follows. This leads us to the following remark.

Remark 2. Fast implementations, such as the one presented in [8], follow our proved restrictions on the number of participants in the service.

References

1. Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006. Preliminary version appeared in PODC 2004.
2. Marcos Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. In *Proceedings of the twenty-eight annual ACM symposium on Principles of distributed computing (PODC09)*, pages 17–25, 2009.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.
4. E. Burkhard, C. Georgiou, P. Musial, N. Nicolaou, and A. A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory, 2009. http://www.cse.uconn.edu/ ncn03001/pubs/TRs/EGMNS09TR.pdf.
5. Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. Parallel Distrib. Comput.*, 69(1):100–116, 2009.
6. Gregory Chockler, Idit Keidar, Rachid Guerraoui, and Marko Vukolic. Reliable distributed storage. *IEEE Computer*, 2008.
7. S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, 2005.
8. Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam 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.
9. Burkhard Englert and Alexander A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, pages 454–463, 2000.
10. Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *Proceeding of the 17th International Symposium on Distributed Computing (DISC)*, pages 75–91, 2003.
11. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
12. Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.
13. Chryssis Georgiou, Peter M. Musial, and Alex A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science*, 383(1):59–85, 2007.

14. Chryssis Georgiou, Peter M. Musial, and Alexander A. Shvartsman. Developing a consistent domain-oriented distributed object service. *IEEE Transactions of Parallel and Distributed Systems, to appear*, 2009. Preliminary version appeared in NCA 2005.
15. Chryssis Georgiou, Nicolas Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi)fast quorum-based implementations of atomic shared memory. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, pages 289–304, 2008.
16. Chryssis Georgiou, Nicolas Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations for atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. Preliminary version appeared in SPAA 2006.
17. David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.
18. S. Gilbert, N. Lynch, and A.A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 259–268, 2003.
19. Vincent Gramoli, Emmanuelle Anceaume, and Antonino 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.
20. Rachid Guerraoui and Marko 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.
21. Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2007.
22. Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
23. Leslie Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.
24. N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, 2002.
25. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
26. Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.
27. Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.
28. Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213, 1998.
29. D. Peleg and A. Wool. Crumbling walls: A class of high availability quorum systems. In *Proceedings of 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 120–129, 1995.
30. Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.