Fault-Tolerant SemiFast Implementations of Atomic Read/Write Registers *

Chryssis Georgiou † Nicolas C. Nicolaou ‡ Alexander A. Shvartsman ‡ §

December 11, 2008

Abstract

This paper investigates time-efficient implementations of atomic read-write registers in message-passing systems where the number of readers can be unbounded. In particular we study the case of a single writer, multiple readers, and S servers, such that the writer, any subset of the readers, and up to t servers may crash. A recent result of Dutta et al. [3] shows how to obtain *fast implementations* in which both reads and writes complete in *one* communication round-trip, under the constraint that the number of readers is less than $\frac{S}{t} - 2$, where $t < \frac{S}{2}$. In that same paper the authors pose a question of whether it is possible to relax the bound on readers, and at what cost, if *semifast* implementations are considered, i.e., implementations that have fast reads or fast writes.

This paper provides an answer to this question. It is shown that one can obtain implementations where all writes are fast, i.e., involving a single communication round-trip, and where reads complete in one to two communication round-trips under the assumption that no more than $t < \frac{S}{2}$ servers crash. Simulated scenarios included in this paper indicate that only a small fraction of reads require a second communication round-trip. Interestingly the correctness of the implementation does not depend on the number of concurrent readers in the system. The solution is obtained with the help of non-unique *virtual ids* assigned to each reader, where the readers sharing a virtual id form a *virtual node*. For the proposed definition of semifast implementations it is shown that implementations satisfying certain assumptions are semifast if and only if the number of virtual ids in the system is less than $\frac{S}{t} - 2$. This result is proved to be tight in terms of the required communication. It is shown that only a *single complete* two communication round-trip read operation may be necessary for each write operation. It is furthermore shown that no semifast implementation exists for the multi-reader, multi-writer model.

Keywords: Fault-tolerance, Distributed algorithms, Atomicity, Read/Write registers, Communication rounds.

Contact Author: Chryssis Georgiou, chryssis@cs.ucy.ac.cy
Tel.: +357 22892745, Fax: +357 22892701

*This work is supported in part by the NSF Grants 9988304, 0121277, and 0311368. A preliminary version of this paper has appeared in [5].

†Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus. Email: chryssis@cs.ucy.ac.cy. The work of this author is supported in part by research funds at the University of Cyprus.

‡Department of Computer Science and Engineering, University of Connecticut, Storrs CT 06269, USA. Email: {nicolas, aas}@engr.uconn.edu.

§Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA.

1 Introduction

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 this 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 communication round-trip, while the read protocol involves two communication round-trips, where the second round-trip essentially performs the write of the value obtained in the first round-trip. Following this development, a folklore belief developed that in message-passing implementations of atomic memory "reads must write". However, recent work by Dutta *et al.* [3] established that if the number of readers is appropriately constrained with respect to the number of object replicas, then single communication round implementations of reads are possible. Such an implementation given in [3] is called *fast*. Furthermore it was shown that any implementation with a large, unconstrained set of readers cannot have only the single round-trip reads. Thus when the number of readers can be large, it is interesting to consider *semifast* implementations where the writes involve a single communication round and where the reads may involve one or two rounds with the goal of having as many as possible single round reads. We note that a communication round-trip involves two communication steps, and henceforth we refer to a communication round-trip as a communication round (defined formally in Section 2.2).

1.1 Background Details

The implementation of atomic SWMR objects in [1] uses *value-timestamp* pairs to impose a partial order on read and write operations. To perform a write operation, the writer increments its local timestamp and sends a message with the value-timestamp pair to all processes. When a majority of processes reply, the write completes. The process performing a read operation sends out queries and waits for a majority of the processes to reply with their value-timestamp pairs. When a majority of the processes replies, the reader finds the highest timestamp and sends the pair consisting of this timestamp and its associated value to all processes. The read completes when the reader receives acknowledgments from a majority of processes. Although the value of the read is established after the first communication round, skipping the second round may lead to violations of atomicity when read operations are concurrent with a write.

Subsequent works extended the approach in [1] to multiple writers, each involving a two round-trip communication protocol, and using quorums of replicas instead of majorities [10, 4]. A fully dynamic atomic memory implementation using reconfigurable quorums is given in [9], where the sets of object replicas can arbitrarily change

over time as processes join and leave the system. When the set of replicas is not being reconfigured, the read and write protocols involve two communication rounds. Retargetting 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. Interestingly, in this work some reads involve a single communication round when it is confirmed that the previous write of the value obtained by the read has already completed.

The implementation of atomic SWMR objects in [3] assumes asynchronous message-passing systems with reliable channels. Here read and write operations are *fast*, i.e., involve a single communication round, but under the constraint that $R < \frac{S}{t} - 2$, where S is the number of servers maintaining object replicas, R is the number of readers, such that the writer, any subset of readers, and up to t servers may crash. The general scheme of the algorithm follows the write operation and the value-timestamp pair technique of [1]. The major departure appears in the execution of a read operation: to decide on the latest value of the replicated object, the read utilizes a predicate on the number of replicas that maintain the maximum timestamp and the number of readers that witnessed the maximum timestamp. Note that for any number $t \geq 1$ of failures the number of readers must be strictly less than the number of servers, and the number of readers is inversely proportional to the number of server failures. A fast implementation cannot exist in the case of multiple readers and multiple writers. For example, it is shown that in the setting where 2 writers and 2 readers exist in the system and $t = 1$, atomicity can be violated.

1.2 Our Contributions

Our goal is to develop atomic memory algorithms where a large number of read and write operations are fast, i.e., involving a single communication round. In particular, we want to remove constraints on the number of readers (with respect to the number of replicas) while preserving atomicity.

We say that an atomic SWMR implementation is *semifast* if write operations take a single communication round and where read operations take one or two rounds. We show that one can obtain semifast implementations with unbounded number of readers, where in many cases reads take a single round. Our approach is based on forming groups of processes where each group is given a unique virtual identifier. The algorithm is patterned after the general scheme of the algorithm in [3]. We show that for each write operation at most one complete read operation returning the written value may need to perform a second communication round. Furthermore, our implementation enables non-trivial executions where both reads and writes are fast, i.e., involve a single communication round. We also provide simulation results for our algorithm, and we consider semifast implementations for multiple writers. In more detail, our contributions are as follows.

1. We define the notion of a *semifast* implementation of atomic objects that specifies what read operations are required to be fast and what read operations are allowed to perform a second communication round (Definition 2.3). In particular, a read operation must be fast if it *precedes* or *succeeds* a *complete* read

operation that performs two communication rounds, when both reads return the value written by the same write operation. The read operations concurrent with a "slow" read operation, may or may not be fast.

2. We provide a semifast implementation of an atomic read/write object that supports arbitrarily many readers (Implementation SF). To accommodate arbitrarily many readers, we introduce the notion of *virtual identifiers* and allow multiple readers to share the same virtual identifier, thus forming groups of nodes that we call *virtual nodes*. We base the determination of the proper return value on the cardinality of the set of virtual nodes maintained by the servers (this is similar to the algorithm in [3] that uses the cardinality of the set of the readers maintained by the servers to determine the return value) . We prove the correctness (atomicity) of the new implementation (Theorem 4.11) and we show that it is indeed semifast (Theorem 5.3). We note that our implementation is not a straightforward extension of [3]. The introduction of virtual nodes raises new challenges such as ensuring consistency within groups so that atomicity is not violated by processes sharing the same virtual id, and proving the resulting implementation correct.

3. We consider two families of algorithms, one that does not use reader grouping mechanisms, the other that assumes grouping mechanisms such as our algorithm. For both we show that there is no semifast atomic implementation if $\frac{S}{t} - 2$ or more virtual identifiers (groups) exist in the system. Additionally it is shown that any semifast algorithm must inform no less than $3t + 1$ server processes during a second communication round (Theorem 6.3).

4. We show that there does not exist semifast atomic implementations for multiple writers and multiple readers, even for $t = 1$ (Theorem 7.2).

5. We simulated our SWMR implementation and we present sample results demonstrating that only a small fraction of read operations need to perform a second communication round. Specifically, under reasonable execution conditions in our simulations no more than 7.5% of the read operations required a second round.

1.3 Paper Organization

The paper is organized as follows. In Section 2 we present our model and definitions (including the formal definition of *semifast* implementation). In Section 3 we describe our SWMR semifast implementation, in Section 4 we prove that it implements atomic read/write registers, and in Section 5 we show that our implementation is indeed semifast. In Section 6 we show the necessary properties that a semifast implementation must possess in order to correctly implement atomic registers. In Section 7 we show that no semifast MWMR implementation is possible. Section 8 contains simulation results. We conclude in Section 9.

2 Model and Definitions

We consider the single writer, multiple reader (SWMR) model. The single writer is a distinguished process w. There are R readers that are processes with unique ids from the set $\mathcal{R} = \{r_1, \ldots, r_R\}$. The writer and any subset of the readers may crash. The object is replicated at S servers with unique ids from the set $\mathcal{S} = \{s_1, \ldots, s_S\}$. Any proper subset of $\mathcal{S}$ of at most t servers can crash ($t < \frac{S}{2}$). A *virtual node* is an abstract entity that consists of a group of reader processes. Each virtual node has a unique identifier from the set $\mathcal{V} = \{\nu_1, \ldots, \nu_V\}$, where $V < \frac{S}{t} - 2$. A reader r_i that is a member of a virtual node ν_j maintains its own identifier r_i and its virtual identifier $\nu(r_i) = \nu_j$; we identify such process by the pair $\langle r_i, \nu_j \rangle$. The processes that share the same virtual identifier are called *siblings*. We assume that some external service is used to create virtual nodes by assigning virtual ids to reader processes. (Note that when $V = R$ and when each virtual node consists of a single unique reader, then our model is essentially that of [3].) We point out that creation of virtual nodes can be accomplished by a local computation. This is due to the fact that the reader participants are not required to have knowledge of their siblings or the membership of other virtual nodes. Thus, a simple inexpensive assignment of virtual ids to nodes may utilize the nodes' own identifiers. In particular, each node r_i can use the knowledge of S, t and $V = \frac{S}{t} - 3$, and use modulo arithmetic to compute $v(r_i) = r_i \mod V$ (this is what we employ in our algorithm and simulation). Observe that a uniform distribution of the readers in the virtual nodes will be achieved using this technique.

Each process p is associated with an application. The application asks the process to invoke an operation and the process responds to the application with the result. We assume a reliable channel between any two processes and that the messages carry a source and a destination field. The state of all channels is represented by the set $mset$ that contains all messages sent but not yet delivered; such messages are said to be *in transit*.

An algorithm A is a collection of automata, where A_p is the automaton assigned to the process p with an initial state $Init_p$. Computation of A proceeds in *steps* where each step denotes actions of a single process. In particular each step is described by an ordered tuple

$$\langle st, p, mIn, inv, mOut, res \rangle$$

where st is the state of the system, p is the process id, mIn the messages received by the process p in that step, inv the invocation submitted to process p by the application, $mOut$ the output messages of process p, and res is the response of the process to the application in that step. The state st includes the set of messages $mset$ and the state of each process i in the system, denoted by st_i. When $inv = \perp$ there is no invocation at that step and when $res = \perp$ there is no response to the application. When $mIn = \emptyset$ or $mOut = \emptyset$ then there are no messages to be received or to be sent out in that step respectively.

For a process p and a step $\langle st, p, mIn, inv, mOut, res \rangle$, the *next state* st' is determined through the following actions: (0) st' is set to st, (1) $st'.mset$ is set to $st.mset - mIn$, (2) process p inputs mIn, inv, and its current

state st_p to A_p, which outputs a new state to st'_p, the messages $mOut$ to be sent, and the response res to the last invoked operation, and (3) p adopts the state st'_p as its new state, sets $st'.mset$ to $st'.mset \cup mOut$, and responds with res to the application.

The step $\langle st, p, mIn, inv, mOut, res\rangle$ is an *invocation step* if $inv \neq \perp$, it is a *response step* if $res \neq \perp$, and a *communication step* if $mOut \neq \emptyset$ and both inv and res are equal to $\perp$.

An *execution fragment* φ of an algorithm A is a finite or infinite sequence of steps $\sigma_0, \sigma_1, \ldots, \sigma_r, \ldots$ of A. An execution fragment is called an *execution* of A if the state in the first step is $s0$, the initial state of the system, where $s0.mset = \emptyset$, and for each process i, $s0_i = Init_i$. We use the symbol ξ to denote executions. We say that an execution fragment φ' *extends* some finite execution fragment φ if the last step of φ is $\sigma = \langle st, \_, \_, \_, \_, \_\rangle$, and if the first step in φ' is $\sigma' = \langle st', \_, \_, \_, \_, \_\rangle$, such that st' is the next state following σ.

A process can crash during any step of an execution. Following a crash the process does not perform any steps. A process is considered to be *faulty* in execution ξ if it crashes in ξ; otherwise the process is *correct*.

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 the value v, initially $\perp$, and the associated timestamp ts, initially 0. A read or a write operation at an application consists of an invocation step and a matching response step. An operation is *incomplete* in an execution, if the operation's invocation step does not have a matching response step; otherwise the operation is *complete*. We assume that application executions are *well-formed* in that it invokes one operation at a time: it waits for a response before invoking another 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 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 failures are constrained by the stated failure model. Atomicity is defined as follows [8]: Consider the set Π of all complete operations in any well-formed execution. Then for operations in Π there exists an irreflexive partial ordering $\prec$ satisfying the following: (1) For any operation π, there are finitely many operations π' such that $\pi' \prec \pi$. (2) If for operations π_1 and π_2, $\pi_1 \rightarrow \pi_2$, then it cannot be the case that $\pi_2 \prec \pi_1$. (3) If π is a write operation and π' is any operation, 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 Semifast Implementations

We want to define a read or write operation π to be *fast* if it completes in one communication round.

Definition 2.1 *A process p performs a **communication round** during operation π in an execution if all of the following hold:*
(1) p sends the messages $m \in mOut$, during the invocation step of operation π or a communication step during π, to a subset of processes,
(2) any process p' that receives $m \in mIn$ from p during a step σ, replies to p with a message $m' \in mOut$ within the same step,[1]
(3) when p receives at least one reply message $m' \in mIn$, it either performs a response step for π or inserts a set of messages in $mOut$ and performs a communication step.

We now formally define fast operations and implementations.

Definition 2.2 *Let π be an operation invoked at process p by an application. If p responds to the application within the first communication round following the invocation of π (in (3) of the above definition), then we say that π is **fast**. An implementation of an atomic object is **fast** if all read and write operations are fast in every execution.*

A semifast atomic implementation, as suggested in [3], is the implementation that either has all reads that are fast *or* all writes that are fast. Here we formalize the notion of semifast implementations. We use the reading-function $\Re(\rho)$ [11] that specifies the (always unique) write operation that writes the value returned by read ρ.

Definition 2.3 *An implementation I of an atomic object is **semifast** if the following are satisfied:*
(1) In any execution ξ of I, every write *operation is fast.*
(2) In any execution ξ of I, any complete read *operation performs one or two communication rounds between the invocation and response.*
(3) For any execution ξ of I, if ρ_1 is a two-round read operation, then any read operation ρ_2 with $\Re(\rho_1) = \Re(\rho_2)$, such that $\rho_1 \rightarrow \rho_2$ or $\rho_2 \rightarrow \rho_1$, must be fast.
(4) There exists an execution ξ of I which contains at least one write operation ω, and at least one read operation ρ_1 which is concurrent with ω and $\Re(\rho_1) = \omega$, such that all read operations ρ with $\Re(\rho) = \omega$ (including ρ_1) are fast.

Notice that property (4) of the above definition requires at least a single fast read operation to be concurrent with the write operation. So trivial solutions that achieve fast operations only in the absence of read and write

[1] Notice that process p' replies to p either at the same step σ or during a subsequent step σ', if p' does not receive any messages between σ and (inclusively) σ'. Intuitively this property is used to forbid processes to wait for other messages before replying to p.

concurrency cannot satisfy property (4). In implementations of atomic objects, we refer to the messages that contain a value to be written to the object as WRITE messages, and we call the messages that request the value of the object as READ messages. The messages used to propagate information within the system are called INFORM messages.

Communication Scheme: We make the following observations in light of Definition 2.3. Given that any subset of the readers and the writer may fail, in order to guarantee termination, we cannot allow any reader or the writer process to wait for replies from any other such process during a read or a write operation. Since we require that the writes are fast, the servers cannot wait for any messages before replying to a WRITE message. Read operations on the other hand are allowed to perform two communication rounds. Two-round reads can have one of the two forms: (i) the reader process may contact the servers twice, (ii) the reader may send messages to the servers during the first round, the servers perform a communication step and contact other servers in the second round and then reply to the reader ending the first round. If the servers are responsible for the second communication round, then it may be the case that all read operations need two rounds to complete, violating semifast properties (3) and (4). Worse yet, a server may fail during its second round preventing an operation from completing. Hence both communication rounds must be performed by the reader when it decides it is necessary to do so according to the information gathered during the first round. Thus from now on we assume that the servers in the semifast implementation, upon receiving a READ or INFORM message, cannot wait for messages from any other process before replying. (Alternatively we can construct executions where only the READ, WRITE and INFORM messages from the invoking processes to the servers and the replies from the servers are delivered. All the other messages remain in transit.)

3 Description of Implementation SF

We now present a semifast SWMR atomic object implementation, called SF, that supports arbitrarily many readers. We assume that the number V of unique virtual ids is such that $V < \frac{S}{t} - 2$. (We show in Section 6 that semifast implementations are impossible when $V \geq \frac{S}{t} - 2$.) Recall that each replica consists of a value and its associated timestamp. For simplicity we give the algorithm that returns only the timestamps; then we describe a straightforward modification that returns the value along with each timestamp. The pseudocode of the implementation is given in Figure 1; line numbers throughout this section refer to this figure. Also in the same figure, the fields that are not required are represented by the "*" symbol in the messages.

Briefly, the write protocol involves the increment of the timestamp and its propagation to all the servers. The operation completes once the writer receives $S-t$ replies from the servers. The read protocol is more complicated. A reader sends read messages to all the servers and once it receives $S-t$ replies, determines the value to be returned

by consulting the validity of a certain predicate. The predicate considers (i) the maximum timestamp witnessed within the replies, (ii) the number of servers that replied with that timestamp, and (iii) the number of virtual nodes whose members witnessed that timestamp through those servers. The idea behind the predicate is presented in detail later in this section. If the predicate holds then the reader returns the maximum timestamp ($maxTS$); otherwise it returns the previous timestamp ($maxTS - 1$). Each server process maintains an object replica and updates its object value when it receives a message that contains a timestamp greater than its local timestamp. The server also records the virtual nodes that requested its object and replies with the information about the object (timestamp,value) along with the recorded set of virtual nodes. Notice that by the read predicate this information is essential for the determination of the value of the atomic object.

Before proceeding to the description of the algorithm we first introduce some notation we use throughout this section and the rest of the paper. For the writer we denote by ω_k the k^{th} write operation and by $\mathcal{S}_{\omega_k}$ the set of servers that received messages from the writer during ω_k. Each read operation is denoted by ρ_i. We say that a read operation ρ_i is invoked by the reader $\langle r_j, \nu_k \rangle$, where r_j is the identifier and ν_k the virtual identifier of the reader. For each read operation ρ_i, let $\mathcal{S}_{\rho_i}$ denote the set of servers that received messages from the process $\langle r_j, \nu_k \rangle$ that invoked ρ_i and replied to those messages. Furthermore let $MaxS_{\rho_i}$ be the set of servers that replied with the maximum timestamp for ρ_i, and therefore $MaxS_{\rho_i} \subseteq \mathcal{S}_{\rho_i}$. The set of messages received from $\langle r_j, \nu_k \rangle$ for ρ_i containing the maximum timestamp and sent by the servers in $MaxS_{\rho_i}$, is represented by MS_{ρ_i}. The maximum timestamp received by $\langle r_j, \nu_k \rangle$ for the read ρ_i is represented as TS_{ρ_i}. If a read operation ρ_i performs a second communication round, then we denote as NS_{ρ_i} to be the set of servers that received the messages from the second communication round of ρ_i and replied to those messages. In this case we say that ρ_i *informs* the servers in NS_{ρ_i}. Lastly for a process p we denote as ts_p the value of the timestamp of p and as $postit_p$ the value of the postit variable at p.

The Writer. The writer w maintains the timestamp and it performs a write operation as follows. It sends a WRITE message consisting of its current timestamp to all the servers (line 8). Since t of the servers might be faulty, w waits for responses from any $S - t$ servers (line 9). Upon receipt these acknowledgments the writer increases its timestamp and completes the operation (lines 9-10). The timestamps impose a natural order on the writes since there is only one writer.

Servers. Each server maintains a replica of the object; this is represented by the object timestamp. The state of a server includes the following variables: (1) ts, the greatest timestamp received by the server, (2) the set *seen* where the server records the *virtual ids* of the readers that inquired about the latest timestamp of the server, (3) the array *counter* of naturals, used to distinguish fresh messages from stale messages from each process (due to asynchrony messages may arrive out of order), and (4) the variable *postit*, used by readers to inform, if necessary, other readers

```
1: at the writer w
2: Components:
3:     ts ∈ ℕ⁺, wCounter ∈ ℕ⁺, v ∈ U
4: procedure initialization:
5:     ts ← 1, wCounter ← 0
6: procedure write(v)
7:     wCounter ← wCounter + 1
8:     send(WRITE, ts, wCounter, 0) to all servers
9:     wait until receive(WRITEACK, ts, *, wCounter, *) from S − t servers
10:    ts ← ts + 1                                              /* reserve a new timestamp * /
11:    return(OK)
12:
13: at each reader r_i
14: Components:
15:    ts ∈ ℕ⁺, maxTS ∈ ℕ⁺, maxPS ∈ ℕ⁺, rCounter ∈ ℕ⁺, v ∈ U
16:    rcvMsg ⊆ M, maxTSmsg ⊆ M, maxPSmsg ⊆ M
17: procedure initialization:
18:    vid(r_i) ← r_i mod (S/t − 3), ts ← 0, rCounter ← 0, maxTS ← 0, maxPS ← 0   /* initialize the virtual ID and other params * /
19:    rcvMsg ← ∅, maxTSmsg ← ∅, maxPSmsg ← ∅
20: procedure read()
21:    rCounter ← rCounter + 1
22:    ts ← maxTS
23:    send(READ, ts, rCounter, vid(r_i)) to all servers
24:    wait until receive(READACK, *, *, Counter, *) from S − t servers s.t. Counter = rCounter
25:    rcvMsg ← {m | r_i received m = (READACK, *, *, rCounter, *)}
26:    maxTS ← max{ts' | (READACK, ts', *, rCounter, *) ∈ rcvMsg}
27:    maxTSmsg ← {m | m.ts = maxTS and m ∈ rcvMsg}             /* gather rcvd messages that contain maxTS * /
28:    maxPS ← max{postit | (READACK, *, *, rCounter, postit) ∈ rcvMsg}
29:    maxPSmsg ← {m | m.postit = maxPS and m ∈ rcvMsg}
30:    for α = 1 to V + 1 do                                    /* look for the min α that satisfies the predicate * /
31:       if there is MS ⊆ maxTSmsg s.t. (|MS| ≥ S − αt) and (| ∩_{m∈MS} m.seen| ≥ α) then   /* check validity of the predicate * /
32:          if | ∩_{m∈MS} m.seen| = α and (maxPS < maxTS or |maxPSmsg| < t + 1) then
33:             send(INFORM, maxTS, rCounter, vid(r_i)) to 3t + 1 servers
34:             wait until receive(INFORMACK, *, *, rCounter, *) from 2t + 1 servers
35:          end if
36:          return(maxTS)
37:          exit procedure read()                             /* return maxTS and exit * /
38:       end if
39:    end for
40:    if maxPS = maxTS then                                    /* α not found so we check the postit * /
41:       if |maxPSmsg| < t + 1 then                            /* proceed to a 2^nd comm. round if "few" postits found * /
42:          send(INFORM, maxTS, rCounter, vid(r_i)) to 3t + 1 servers
43:          wait until receive(INFORMACK, *, *, rCounter, *) from 2t + 1 servers
44:       end if
45:       return(maxTS)
46:       exit procedure read()
47:    else
48:       retutn(maxTS − 1)
49:       exit procedure read()                                 /* if neither α or postits = maxTS found return maxTS − 1 * /
50:    end if
51:
52: at each server s_i
53: Components:
54:    ts ∈ ℕ⁺, counter[0...R] ∈ ℕ⁺, v ∈ U, postit ∈ ℕ⁺
55:    msgType ∈ {WRITE,READ,INFORM}, seen ⊆ 𝒱 ∪ {w}
56: procedure initialization:
57:    ts ← 0, seen ← ∅, counter[0...R] ← 0, postit ← 0
58: procedure serve()
59:    upon receive(msgType, ts', rCounter', vid) from q ∈ {w, r_1, ..., r_R} and rCounter' ≥ counter[pid(q)] do
60:    if ts' > ts then                                         /* update local timestamp and seen sets as necessary * /
61:       ts ← ts'; seen ← {vid};
62:    else
63:       seen ← seen ∪ {vid}
64:    end if
65:    counter[pid(q)] ← rCounter'                              /* pid(q) returns 0 if q = w and i if q = r_i * /
66:    if msgType =READ
67:       send(READACK, ts, seen, rCounter', postit) to q
68:    else if msgType =WRITE
69:       send(WRITEACK, ts, seen, rCounter', postit) to q
70:    else if msgType =INFORM
71:       if postit < ts' then
72:          postit ← ts'                                       /* update postit value if necessary * /
73:       end if
74:       send(INFORMACK, *, *, rCounter', postit) to q
75:    end if
```


Figure 1: Implementation SF.

about the timestamp they are about to return.

We now describe the operation of a server s_i when it receives a message $(msgType, ts', rCounter', vid)$ from a non-server process p_j (line 59). Upon receipt of this message, server s_i updates its timestamp ts if $ts' > ts$ and initializes its *seen* set to $\{\nu(p_j)\}$, the virtual id of p_j (lines 60-61). Otherwise, if $ts' \leq ts$, s_i sets its *seen* set to be equal to $seen \cup \{\nu(p_j)\}$ (line 63) declaring that p_j inquired s_i's timestamp. This is a departure from the algorithm in [3]: we record the virtual identifier of p_j, using its unique identifier only for message exchange. By doing so we manage to keep $|seen| < \frac{S}{t} - 2$ (required for correctness) without having to bound the number of readers. Server s_i then sends an appropriate reply to p_j acknowledging the request. If a READ, WRITE or INFORM message was received then the reply is a READACK, WRITEACK, or an INFORMACK, respectively (lines 66-75).

Receiving an INFORM message denotes that process p_j wants to inform the rest of the reader processes about the timestamp it is about to return. Before replying to such message, s_i updates its *postit* value, provided the received timestamp ts' is greater than *postit* (lines 71-73). Otherwise *postit* is not updated, since ts' is an "older" timestamp than *postit* (updating *postit* with an older timestamp may lead to violation of atomicity).

Along with any reply, s_i encloses its timestamp ts, its *seen* set, its *counter*, and the value of *postit*.

Reader. The actions of a reader node with id r_i and virtual id $\nu(r_i) = \nu_j$ are as follows (lines 13-50). Each reader maintains the following state: (1) variable $maxTS$, which holds the maximum timestamp that a reader received from the servers during the reader's last read operation, (2) variable $rCounter$ to count the number of read operations performed by the reader (used to distinguish fresh messages from stale messages from the reader), (3) variable $maxPS$ that holds the maximum $postit$ value that the reader witnessed among the responses.

The reader r_i performs a read operation as follows. Upon invocation, it sends messages to all servers and waits for $S - t$ responses (lines 23-24). Each of these responses is of the form (READACK, $ts', seen, Counter, postit$). While collecting these messages, r_i checks $Counter$ to distinguish fresh messages (with $Counter = rCounter$) from stale messages, and then records the maximum timestamp $maxTS = ts'$ (line 26) and the maximum postit $maxPS = postit$ value (line 28) contained among the received messages. Based on the received information, reader r_i computes the set of messages $maxTSmsg$ that contained the maximum timestamp (line 27). Then a key predicate (line 31) is used to decide the return value: the reader searches for the minimum $\alpha \in [1, V + 1]$ such that there exists a subset $MS \subseteq maxTSmsg$ with cardinality $|MS| \geq S - \alpha t$, and the cardinality of the intersection of the seen sets of the messages in MS is $|\cap_{m \in MS} m.seen| \geq \alpha$. If such an α exists then the reader returns $maxTS$. Else it returns $maxTS - 1$. This predicate, informally, asks the following question: "have enough processes seen $maxTS$ timestamp that the reader received?"

In order to visualize the idea behind the predicate consider a finite execution fragment φ_1 where the writer performs a complete write operation ω that receives replies from the set $\mathcal{S}_\omega$ of servers, such that $|\mathcal{S}_\omega| = S - t$. We extend φ_1 by a complete read operation ρ_1 that misses t servers from those that responded in ω, such that

$|MS_{\rho_1}| = |\mathcal{S}_\omega \cap \mathcal{S}_{\rho_1}| = S - 2t$, where $\mathcal{S}_{\rho_1}$ is the set of $S - t$ servers that responded in ρ_1. Atomicity requires ρ_1 to return $TS_{\rho_1} = maxTS$. Consider now another execution φ_2 where the write operation ω is incomplete and receives replies from exactly $|\mathcal{S}_\omega| = S - 2t$ servers. We extend φ_2 with a read operation ρ_1 at r_i that receives replies from $|\mathcal{S}_{\rho_1}| = S - t$ servers, including the servers in $\mathcal{S}_\omega$. So $|MS_{\rho_1}| = |\mathcal{S}_\omega \cap \mathcal{S}_{\rho_1}| = S - 2t$ and thus ρ_1 cannot distinguish execution φ_2 from φ_1. Hence, by atomicity, ρ_1 must return $TS_{\rho_1} = maxTS$ in φ_2 as well. By extending φ_2 even further by a second read operation ρ_2 at r_j we might get into the situation where $|MS_{\rho_2}| = |\mathcal{S}_{\omega_1} \cap \mathcal{S}_{\rho_2}| = S - 3t$, with $|\mathcal{S}_{\rho_2}| = S - t$ the servers that responded in ρ_2. But in order to preserve atomicity, the reader r_j must also return $TS_{\rho_2} = maxTS$. This scenario can be generalized for more than two read operations and so the predicate in line 31 of the algorithm in Figure 1 serves to preserve atomicity of the different read operations.

Note here that the above scenario assumes the recording of the unique ids of the readers in the *seen* set. Our approach is for the servers to record the virtual ids of the readers. So it is possible that after two subsequent read operations the cardinality of the seen set remains the same. Hence in the above scenario, if both r_i and r_j belong in the same virtual node ν_k, then for ρ_1 and ρ_2 in φ_2 it is the case that $|\cap_{m \in MS_{\rho_1}} m.seen| = |\cap_{m \in MS_{\rho_2}} m.seen| = |\{w, \nu_k\}| = 2$. In this case the predicate holds for ρ_1, so it returns $maxTS$, but it does not hold for ρ_2, and if it returns $maxTS - 1$ it would violate atomicity.

A second communication round is necessary when r_i satisfies the predicate such that $|\cap_{m \in MS} m.seen| = \alpha$. During the second communication round, r_i informs $3t + 1$ servers about the timestamp it is about to return. Since t servers might be faulty, r_i completes as soon as it receives $2t + 1$ acknowledgments and returns $maxTS$.

In the case where the predicate is false, reader r_i checks if there was any *postit* equal to $maxTS$ observed, as advertised within the received messages. If so, then some reader (previously or concurrently with r_i) returned or is about to return $maxTS$. If r_i receives more than $t + 1$ messages containing that postit, it returns $maxTS$ without performing a second communication round; otherwise a second communication round is required by r_i to ensure that any subsequent reader will receive the same postit. If neither $postit$ equals $maxTS$, then r_i returns $maxTS - 1$ in one communication round.

Returning values with timestamps. A slight modification needs to be applied to the algorithm to associate returned timestamps with values. To do this the writer attaches two values to the timestamp in each write operation: (1) the current value to be written, and (2) the value written by the immediately preceding write operation (for the first write this is $\perp$). The reader receives the timestamp with its two attached values. If, as before, it decides to return $maxTS$, then it returns the current value attached to $maxTS$. If the reader decides to return $maxTS - 1$, then it returns the second value (corresponding to the preceding write).

4 SF Implements an Atomic Read/Write Register

We now prove the correctness of algorithm SF. Generally speaking processes can fail in any stage of their execution. We do not assume that in algorithm SF (Figure 1) the lines of code are executed atomically: processes may crash in the middle of a line or between the lines. In particular, while sending messages to a set of processes, the sending process may crash after sending messages to an arbitrary subset (however each message is sent in its entirety). In the rest of the section we use the notation presented in Section 3.

Since the correctness of our implementation depends mainly on the timestamps written and returned, we reduce the properties of the atomicity presented in Section 2, to the following:

A1: If a read operation returns, it returns a non-negative integer,

A2: if a read ρ is complete and succeeds some write ω_k, then ρ returns ℓ such that $\ell \geq k$,

A3: if a read ρ returns k $(k \geq 1)$, then ω_k either precedes ρ or is concurrent with ρ,

A4: if some read ρ_1 returns k $(k \geq 0)$ and a read ρ_2 that succeeds ρ_1 returns ℓ, then $\ell \geq k$.

We will show that implementation SF preserves each and every of the above conditions in any given execution.

We begin with a lemma that plays a significant role in the correctness of our implementation. The lemma follows from the fact that no more than t servers might fail and that the communication channels are reliable.

Lemma 4.1 *Let two readers with actual ids $\langle r_i, \nu_k \rangle$ and $\langle r_j, \nu_k \rangle$ be siblings and make subsequent reads ρ_1 and ρ_2, respectively. Then, for any execution ξ of SF, $| \, |MaxS_{\rho_1}| - |MaxS_{\rho_2}| \, | \leq t$.*

We now proceed to show atomicity condition **A1**.

Lemma 4.2 *For any execution ξ of SF, if a read operation in ξ returns, it returns a non negative integer.*

Proof. Consider an execution ξ of SF. As previously noted the servers initially in ξ maintain a timestamp $ts = 0$. Consider now a read operation ρ_1 in ξ that is performed by the reader $\langle r_i, \nu_i \rangle$ and precedes any write operation. It follows that ρ_1 will receive $\geq S - t$ replies with $TS_{\rho_1} = 0$. Before replying to ρ_1, each server $s_j \in \mathcal{S}_{\rho_1}$ adds the virtual id, ν_i, of the reader in its seen set. Since all the servers in $\mathcal{S}_{\rho_1}$ will reply with the same timestamp $ts = 0$ and since the seen set of each $s_i \in \mathcal{S}_{\rho_1}$ contains at least the element ν_i, it follows that $MS_{\rho_1} \geq S - t$ and the predicate will be true for $\alpha = 1$. Therefore, ρ_1 will return $TS_{\rho_1} = 0$ which is not negative.

We now consider the case where a write operation precedes the read operation ρ_1 in execution ξ. The writer invokes a $write(x)$ operation where $x \geq 1$ the timestamp to be written. Since $write(x)$ is invoked we have that $write(x-1)$ is completed and so the operation ρ_1 will receive $TS_{\rho_1} = x$ or $TS_{\rho_1} = x - 1$. If $TS_{\rho_1} = x$, ρ_1 will return either $ts = x$ or $ts = x - 1$ and so $ts_{r_i} \geq 0$. If $TS_{\rho_1} = x - 1$, and since $write(x-1)$ is completed,

then at least $|MaxS_{\rho_1}| = S - 2t$ servers would reply to rd_1 with timestamp $ts = x - 1$. Moreover, every server $s_i \in MaxS_{\rho_1}$, received messages from the write operation $write(x-1)$ and thus added the identifier w of the writer in their seen set before replying to the write operation. Also every $s_i \in MaxS_{\rho_1}$ added the virtual id of the reader, ν_i, in their seen set before replying to ρ_1. Since the writer does not have any sibling process we have that $w \neq \nu_i$ and so the cardinality of the seen set of each $s_i \in MS_{\rho_1}$ is greater or equal to 2. Therefore, the predicate will be true for $\alpha = 2$ for ρ_1 and so the read operation will return $ts = x - 1 \geq 0$. This completes the proof. □

We now show that the timestamp of any server is monotonically increasing.

Lemma 4.3 *In any execution ξ of SF, if a server s_i sets its timestamp ts_{s_i} to x at step σ, then, given any step σ' of ξ such that $\sigma < \sigma'$ and $ts_{s_i} = y$, we have that $y \geq x$.*

Proof. This can be ensured by line 44 of implementation SF (Figure 1). If we consider an execution ξ of SF, observe that a server s_i only changes its timestamp ts_{s_i}, if the timestamp received, ts_x, is $ts_x > ts_{s_i}$. If the condition is true then s_i upgrades its timestamp by $ts_{s_i} = ts_x$, otherwise ts_{s_i} remains unchanged. This assures that the timestamp on the server site increases monotonically. □

We now show the monotonicity of the postits for any server.

Lemma 4.4 *In any execution ξ of SF, if a server s_i sets its $postit_{s_i}$ to x at step σ, then, given any step σ' of ξ such that $\sigma < \sigma'$ and $postit_{s_i} = y$, we have that $y \geq x$.*

Proof. This is ensured by line 56 of implementation SF (Figure 1). □

Given the above lemmas we prove the second atomicity property (**A2**):

Lemma 4.5 *For any execution ξ of SF if a read ρ_1 is complete and succeeds some write ω_k ($\omega_k \rightarrow \rho_1$), then ρ_1 returns ℓ such that $\ell \geq k$.*

Proof. Suppose that the writer ω performs a ω_k operation and precedes the read ρ_1 operation by reader $\langle r_i, \nu_i \rangle$ during an execution ξ of SF. Let $\mathcal{S}_{\omega_k}$ be the $S - t$ servers that replied to ω_k in the same execution. The intersection between $\mathcal{S}_{\omega_k}$ and $\mathcal{S}_{\rho_1}$, $MaxS_{\rho_1} = \mathcal{S}_{\omega_k} \cap \mathcal{S}_{\rho_1}$, is obviously $|MaxS_{\rho_1}| \geq S - 2t$. Since $\omega_k \rightarrow \rho_1$ the timestamp ts for each server in $MaxS_{\rho_1}$, per Lemma 4.3, is greater or equal to k. So ρ_1 received a maximum timestamp TS_{ρ_1} such that $TS_{\rho_1} \geq k$. From the implementation we know that the reader returns either TS_{ρ_1} or $TS_{\rho_1} - 1$. We consider two cases:

Case 1: $TS_{\rho_1} > k$. Since ρ_1 returns either TS_{ρ_1} or $TS_{\rho_1} - 1$, it follows that either case it returns a timestamp greater or equal to k.

Case 2: $TS_{\rho_1} = k$. As we mentioned above each server in $MaxS_{\rho_1}$ replies with a $ts \geq k$. Since $TS_{\rho_1} = k$

every server $s_i \in MaxS_{\rho_1}$ replies with a timestamp $ts = k$ to ρ_1. So the set MS_{ρ_1}, which contains the messages received by ρ_1 with the highest timestamp, will include the messages sent by all the servers in $MaxS_{\rho_1}$. So $|MS_{\rho_1}| \geq S - 2t$. But since the writer sent a message with timestamp k to the servers before ρ_1, then w is included in the $seen$ set of each server in $MaxS_{\rho_1}$. Before the servers in $MaxS_{\rho_1}$ responded to ρ_1 they also included ν_i in their $seen$ set. So the predicate will be true for $\alpha = 2$ and ρ_1 will return $TS_{\rho_1} = k$. Observe that any read operation returns TS_{ρ_1}, since the writer w has no sibling, and thus the predicate holds for $\alpha = 2$ no matter which reader performs the read operation. □

We say that a $postit = x$ is *introduced* in the system by a read operation ρ, if ρ is complete, performs two communication rounds, and for every INFORM message (INFORM, ts, _, _) it sends during its second round, $ts = x$. The following lemma ensures that if a $postit = x$ is introduced to the system, then there exists a maximum timestamp ts in the system such that $ts \geq x$.

Lemma 4.6 *For any execution ξ of SF, if a $postit = x$ is introduced in the system by a read operation ρ_1, then any succeeding read operation ρ_2 will observe a maximum timestamp ts' such that $ts' \geq x$.*

Proof. Consider an execution ξ of SF where the read operation ρ_1 introduced a postit equal to x to the system. It follows that ρ_1 observed as the maximum timestamp in the system $TS_{\rho_1} = x$. Assume that $|MS_{\rho_1}| \geq S - \alpha t$ and $|\cap_{m \in MS_{\rho_1}} m.seen| = \alpha$, and thus ρ_1 performs an informative operation. Since $\alpha \in [1, V+1]$ and $S > (V+2)t$, we get that $|MS_{\rho_1}| > t$. So, if we denote by $\mathcal{S}_{\rho_2}$ the set of servers that replied to the succeeding read ρ_2 ($|\mathcal{S}_{\rho_2}| = S - t$), then per Lemma 4.3 there is a server, $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ that replies to ρ_2 with a timestamp $ts' \geq x$. Therefore, ρ_2 detects a maximum timestamp $TS_{\rho_2} \geq ts'$, and hence $TS_{\rho_2} \geq x$. □

Lemma 4.7 *For any execution ξ of SF if a read operation ρ_1 receives a $postit = x$ then ρ_1 will return a value $y \geq x$.*

Proof. Consider an execution ξ of SF which contains a read operation ρ_1 by a reader $\langle r_i, \nu_i \rangle$. It follows from Lemma 4.6 that if read ρ_1 receives a $postit = x$, then it will detect a maximum timestamp $TS_{\rho_1} \geq x$. Let $TS_{\rho_1} = x$ and so either the predicate will hold and then ρ_1 will return $y = TS_{\rho_1}$, or the condition whether $postit_{r_i} = TS_{\rho_1}$ will be true and so ρ_1 will in this case return $y = TS_{\rho_1}$ as well. Thus ρ_1 will return $y = x$. If now $TS_{\rho_1} > x$ then ρ_1 will return $y = TS_{\rho_1}$ if the predicate holds or $y = TS_{\rho_1} - 1$ otherwise. Note that since $postit = x$, it is less than TS_{ρ_1} and so the postit condition does not hold. Either case ρ_1 will return a value $y \geq x$. □

We proceed to the proof of property **A3**.

Lemma 4.8 *In any execution ξ of SF if a read operation ρ_1 returns $k \geq 1$, then the write operation ω_k either precedes ρ_1 ($\omega_k \rightarrow \rho_1$) or is concurrent with ρ_1.*

Proof. Consider an execution ξ of SF. Note that in order for a timestamp $ts = k$ to be introduced in the system during ξ a write operation ω_k must be invoked (since only the writer increments the timestamp). We now investigate what happens when a reader returns a timestamp $ts = k$ in ξ. Let TS_{ρ_1} be the maximum timestamp received by the read operation ρ_1. Then ρ_1 returns, according to the implementation, either $k = TS_{\rho_1}$ or $k = TS_{\rho_1} - 1$. The first case is possible if the predicate holds for the reader or if the reader observed some postit, such that $postit = TS_{\rho_1}$. If the predicate holds then ρ_1 detected timestamp $TS_{\rho_1} = k$ in $|MS_{\rho_1}| \geq S - \alpha t$ messages, and since $V < \frac{S}{t} - 2$ and $\alpha \leq V + 1$ then $|MS_{\rho_1}| > t$. So there is at least one server $s_i \in \mathcal{S}_{\rho_1}$ that received messages from ω_k before replying to ρ_1. If ρ_1 returns k because of a postit, then per Lemma 4.6, timestamp $ts = k$ was already introduced in the system. Thus for both cases ω_k is either concurrent or precedes the read operation ρ_1.

In the case where the reader returns $k = TS_{\rho_1} - 1$ it follows that the reader detected a maximum timestamp $TS_{\rho_1} = k + 1$ in the system and thus the ω_{k+1} operation has already been initiated by the writer. Hence, ω_k operation has already been completed and preceded ρ_1 or was concurrent and completed before ρ_1 completes. □

In order to prove the atomicity property **A4**, we first need to show that readers who belong to the same virtual node (siblings) satisfy the property (Lemma 4.9). Then we show that the property is also true for any two non-sibling readers in the system (Lemma 4.10). The idea behind the proofs is to investigate the possible states of the predicate used for a read operation in SF and show that the property **A4** is not violated by any of them. Since the predicate is affected by the chosen α (number of replicas with the maximum timestamp) and the cardinality and the membership of the intersection of the seen sets, we analyze each case separately.

Lemma 4.9 *Let the readers $\langle r_i, \nu_k \rangle$ and $\langle r_j, \nu_k \rangle$ be siblings and perform the read operations ρ_1 and ρ_2 respectively. For any execution ξ of SF that contains ρ_1 and ρ_2, if $\rho_1 \rightarrow \rho_2$, and ρ_1 returns x then ρ_2 returns y, such that $y \geq x$.*

Proof. We consider again an execution ξ of SF. We first investigate the case where $r_i = r_j$. In this case ρ_1 denotes the first read operation of r_i and ρ_2 a succeeding read operation from the same reader. Let x be the value returned from ρ_1. During the read ρ_2, r_i sends a READ message with $ts_{r_i} = TS_{\rho_1} \geq x$. This message will be received by all servers in $\mathcal{S}_{\rho_2}$ which according to Lemma 4.3 will reply with a timestamp $ts' \geq TS_{\rho_1} \geq x$. So $TS_{\rho_2} \geq x$. If $TS_{\rho_2} = x$ then $|MS_{\rho_2}| = S - t$ and the predicate holds for $\alpha = 1$. Thus $y = TS_{\rho_2} = x$. Otherwise, if $TS_{\rho_2} > x$, the return value y will be equal to TS_{ρ_2} or $TS_{\rho_2} - 1$ and thus $y \geq x$. By a simple induction we can show that this is true for every read operation of r_j(including ρ_2) after ρ_1. For the rest of the proof we assume that $r_i \neq r_j$. We investigate the following two possible cases: (1) ρ_1 returns $x = TS_{\rho_1} - 1$ and (2) ρ_1 returns $x = TS_{\rho_1}$. In all of the cases we show that $x \leq y$ or that the case is impossible.

Case 1: In this case $x = TS_{\rho_1} - 1$. Therefore, some servers replied to ρ_1 with $TS_{\rho_1} = x + 1$, and hence a write operation ω_{x+1} started before ρ_1 is completed. So ω_x completed before ρ_1 has completed and therefore $\omega_x \rightarrow \rho_2$

since $\rho_1 \rightarrow \rho_2$. Thus by Lemma 4.5 ρ_2 returns a value $y \geq x$.

Case 2: In this case $x = TS_{\rho_1}$. Hence either there is some $\alpha \in [1, V+1]$ such that $|MS_{\rho_1}| \geq S - \alpha t$ and $|\cap_{m \in MS_{\rho_1}} m.seen| \geq \alpha$ or ρ_1 received a $postit$ equal to TS_{ρ_1} from some server. We examine those two possibilities separately.

Case 2(a): It follows that $x = TS_{\rho_1}$, and there is some $\alpha \in [1, V+1]$ such that MS_{ρ_1} consist at least $S - \alpha t$ messages received by ρ_1 with $ts = x$ and $|\cap_{m \in MS_{\rho_1}} m.seen| \geq \alpha$. Since $V < \frac{S}{t} - 2$ and $a \in [1, V+1]$, then $|MS_{\rho_1}| = S - at > t$. Following we investigate the cases where $|\cap_{m \in MS_{\rho_1}} m.seen| = \alpha$ and $|\cap_{m \in MS_{\rho_1}} m.seen| > \alpha$. (1) First lets examine the case where ρ_1 returns $x = TS_{\rho_1}$ because $|\cap_{m \in MS_{\rho_1}} m.seen| = \alpha$. According to the implementation, ρ_1 has to inform $|NS_{\rho_1}| \geq 2t + 1$ servers about its return value, x. Since ρ_1 precedes ρ_2, at least $|NS_{\rho_1} \cap \mathcal{S}_{\rho_2}| \geq t+1$ servers, that informed by ρ_1, will reply to ρ_2. Any server $s_i \in NS_{\rho_1} \cap \mathcal{S}_{\rho_2}$, by Lemma 4.6 will reply with a $postit_{s_i} \geq x$ to ρ_2 and with a timestamp $ts_{s_i} \geq x$. So ρ_2 will observe a maximum timestamp $TS_{\rho_2} \geq x$. According now to Lemma 4.7 ρ_2 will return a value $y \geq x$. (2) The second case arise when ρ_1 returns $x = TS_{\rho_1}$ because $|\cap_{m \in MS_{\rho_1}} m.seen| > \alpha$. We can split this case in two subcases regarding the value returned by ρ_2. The two possible values that ρ_2 might return is $y = TS_{\rho_2}$ or $y = TS_{\rho_2} - 1$:

(i) Let first consider the case where $y = TS_{\rho_2}$. Since ρ_1 returned $x = TS_{\rho_1}$, as we showed in lemma 4.8 , there is a write operation ω_x that precedes or is concurrent with ρ_1. As stated above $|MS_{\rho_1}| > t$ and hence there is a server s_i such that $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$. By Lemma 4.3, s_i will send a timestamp $ts \geq x$ to ρ_2, and hence $TS_{\rho_2} \geq ts$. So $y \geq x$.

(ii)We now get down to the case where ρ_2 returns $y = TS_{\rho_2} - 1$. Since $|MaxS_{\rho_1}| > t$, there must be a server $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ and s_i replies with a timestamp $ts \geq x$ to ρ_2. So the highest timestamp in $\mathcal{S}_{\rho_2}$(i.e. $TS_{\rho_2} = y + 1$) will be greater or equal to x. If the inequality is true, namely $y + 1 > x$, then clearly the value returned by ρ_2 is $y \geq x$. If the equality holds and $y + 1 = x$ then the highest timestamp received by ρ_2, $TS_{\rho_2} = y + 1 = x$. Hence all the servers in $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ replied with a timestamp $ts = x = y + 1$ to ρ_2. Recall that in this case we assumed that $|\cap_{m \in MS_{\rho_1}} m.seen| > \alpha$. Also according to Lemma 4.1, $||MS_{\rho_2}| - |MS_{\rho_1}|| \leq t$ and since $|MaxS_{\rho_1}| = |MS_{\rho_1}| \geq S - \alpha t$, it follows that ρ_2 will receive the maximum timestamp $TS_{\rho_2} = x$ from $|MaxS_{\rho_2}| = |MS_{\rho_2}| \geq S - (\alpha + 1)t$ servers. Let for any $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$, denote by m_1 the message sent from s_i to ρ_1 and m_2 the message sent to ρ_2. Obviously $m_1.ts = m_2.ts = x$. Since the timestamp is the same and s_i sent m_1 before m_2 then $m_1.seen \subseteq m_2.seen$. As a result $|\cap_{m \in MS_{\rho_1}} m.seen| \leq |\cap_{m \in MS_{\rho_2}} m.seen|$. Notice that, since the two readers are siblings, if no *non-sibling* reader received replies from those servers in between ρ_1 and ρ_2, then $m_1.seen = m_2.seen$ and $|\cap_{m \in MS_{\rho_1}} m.seen| = |\cap_{m \in MS_{\rho_2}} m.seen|$. Either case, $|\cap_{m \in MS_{\rho_2}} m.seen| > \alpha$ and hence $|\cap_{m \in MS_{\rho_2}} m.seen| \geq \alpha + 1$. Observe that the predicate now is true for $\alpha + 1$ since $|MS_{\rho_2}| \geq S - (\alpha + 1)t$, and thus ρ_2 must return $TS_{\rho_2} = x = y + 1$, contradicting the initial assumption that ρ_2 returns $y = x - 1$. The same result applies in both cases where $\alpha \leq V$ and $\alpha = V + 1$ since the $seen$ set

remains unchanged.

Case 2(b): Here ρ_1 returns $x = TS_{\rho_1}$ because some postits equal to TS_{ρ_1} received by ρ_1. We have to consider 2 cases here. Either (1) ρ_1 received more than $t+1$ postits, or (2) ρ_1 received less than $t+1$ postits. Both cases imply that, a reader $\langle r_m, \nu_n \rangle$ perform a read ρ_{1a}, and is about to return or already returned the maximum timestamp (which is equal to TS_{ρ_1}) in the system. Furthermore implies that ρ_{1a} initiated an informative phase which is concurrent or precedes the read operation ρ_1. By analyzing the cases we obtain the following results:
(1) If ρ_1 received more than or equal to $t+1$ messages containing a postit with value $postit = TS_{\rho_1} = x$, then the writer w initiated a ω_x operation during or before ρ_1 is completed. It follows that $NS_{1a} \cap \mathcal{S}_{\rho_1}$ denote the set of servers that replied to ρ_1 and contained the $postit = TS_{\rho_1}$. The reader ρ_2 receives replies from $|\mathcal{S}_{\rho_2}| = S - t$ servers. Since $|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| \geq t+1$, then $|\mathcal{S}_{\rho_2} \cap (NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1})| \geq 1$. So the read operation ρ_2 will receive a reply from at least one server $s_i \in NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}$. Hence, from Lemma 4.4, ρ_2 receives a $postit_{s_i} \geq x$ from s_i and according to Lemma 4.7 will return a value $y \geq postit_{s_i}$ and thus $y \geq x$.
(2) Let now examine the case where ρ_1 receives less than $t+1$ messages containing postits with value equal to TS_{ρ_1}. Let assume again that $|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| < t+1$ is the set of servers that replied with $postit = TS_{\rho_1}$ to ρ_1. However, in contrary to the previous case, the situation where $|(NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}) \cap S2| = 0$ is possible. So ρ_1 informs $|NS_{\rho_1}| \geq 2t+1$ servers with a $postit = TS_{\rho_1}$ before completing. So there exists a server $s_i \in \mathcal{S}_{\rho_2} \cap NS_{\rho_1}$ that replies to ρ_2. By Lemma 4.4, s_j replies with a $postit_{s_j} \geq TS_{\rho_1}$, and by Lemma 4.7, ρ_2 returns a timestamp $y \geq postit_{s_j}$. Hence ρ_2 returns a value $y \geq x$. □

We now show that the forth atomicity property is preserved by the operations invoked from non-sibling readers.

Lemma 4.10 *Let the readers $\langle r_i, \nu_i \rangle$ and $\langle r_j, \nu_j \rangle$ be non-siblings and perform the read operations ρ_1 and ρ_2 respectively. For any execution ξ of SF that contains ρ_1 and ρ_2, if $\rho_1 \rightarrow \rho_2$, and ρ_1 returns x then ρ_2 returns y, such that $y \geq x$.*

Proof. Consider an execution ξ of SF. In this lemma we study the case where $r_i \neq r_j$ and $\nu_i \neq \nu_j$ in ξ, and hence the two readers are not siblings. We proceed in cases and show that $y \geq x$ or the case is impossible. We know that ρ_1 may return either $TS_{\rho_1} - 1$ or TS_{ρ_1}. It can be shown similarly to case (1) of Lemma 4.9 that when ρ_1 returns $x = TS_{\rho_1} - 1$ then ρ_2 returns $y \geq x$. It remains to investigate the cases where: (1) ρ_1 returns TS_{ρ_1} because the predicate did not hold but it received some postits, such that $postit = TS_{\rho_1}$, and (2) ρ_1 returns TS_{ρ_1} because it received $|MS_{\rho_1}|$ messages that contained the maximum timestamp TS_{ρ_1} such that there is $\alpha \in [1 \ldots V+1]$ and $|MS_{\rho_1}| \geq S - \alpha t$ and $|\cap_{m \in MS_{\rho_1}} m.seen| \geq a$.

Case 1: In this case ρ_1 returns $x = TS_{\rho_1}$ because it received some postits, s.t. $postit = TS_{\rho_1}$. According to the implementation some process (sibling or not of r_i), say r_k, performed a read operation ρ_{1a} and is about, or already returned a timestamp equal to TS_{ρ_1}. There are two cases to consider based on the cardinality of $NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}$: (1)

$|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| \geq t+1$ and (2) $|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| < t+1$. If (1) is true and r_i received $|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| \geq t+1$, then ρ_1 returns $x = TS_{\rho_1}$ without performing a second communication round. Since the set of servers that responded to ρ_2 is $|\mathcal{S}_{\rho_2}| = S - t$, it follows that there is at least one server $s_i \in \mathcal{S}_{\rho_2} \cap (NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1})$. According to Lemma 4.4, s_i will relpy to ρ_2 with a $postit_{s_i} \geq x$. Furthermore by Lemma 4.7, ρ_2 will return a value $y \geq postit_{s_i}$. So obviously ρ_2 returns a value $y \geq x$. On the other hand if (2) is true and ρ_1 received $|NS_{\rho_{1a}} \cap \mathcal{S}_{\rho_1}| < t+1$ postits, then, before returning, ρ_1 will inform $|NS_{\rho_1}| \geq 2t+1$ servers with a $postit = TS_{\rho_1}$. So there exists a server $s_i \in \mathcal{S}_{\rho_2} \cap NS_{\rho_1}$ that replies to ρ_2. By Lemma 4.4 s_i, will reply with a $postit_{s_1} \geq TS_{\rho_1}$ and by Lemma 4.7 it follows that ρ_2 will return a timestamp $y \geq postit_{s_i}$. Hence it follows again that $y \geq x$.

Case 2: This is the case where ρ_1 returns TS_{ρ_1} because the predicate holds, namely, there is $\alpha \in [1 \dots V+1]$ and $|MS_{\rho_1}| \geq S - \alpha t$ such that $|\cap_{m \in MS_{\rho_1}} m.seen| \geq \alpha$. Recall again that since $\alpha \in [1 \dots V+1]$ and $V < \frac{S}{t} - 2$, $|MS_{\rho_1}| \geq S - \alpha t > t$. So, if $MaxS_{\rho_1}$ are the servers that replied with messages in MS_{ρ_1}, there is at least one server $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$. Therefore s_i replies to ρ_2, by Lemma 4.3, with a timestamp $ts \geq x$. Hence ρ_2 will observe a maximum timestamp $TS_{\rho_2} \geq x$. If ρ_2 observes $TS_{\rho_2} > x$ then clearly, since ρ_2 returns either $y = TS_{\rho_2}$ or $y = TS_{\rho_2} - 1$, it will return a value $y \geq x$. It remains to investigate the case where the maximum timestamp observed by ρ_2 is $TS_{\rho_2} = x$. Since $TS_{\rho_2} = TS_{\rho_1} = x$ it follows that all the servers in $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ will reply to ρ_2 with a timestamp $ts = x$. Furthermore, since ρ_2 might miss up to t servers from $MaxS_{\rho_1}$ and $|MaxS_{\rho_1}| = |MS_{\rho_1}| \geq S - \alpha t$, it follows that ρ_2 will receive the maximum timestamp $TS_{\rho_2} = x$ from $|MaxS_{\rho_2}| = |MS_{\rho_2}| \geq S - (\alpha+1)t$ servers. There are two possible return values for ρ_2. Either $y = TS_{\rho_2} = x$ or $y = TS_{\rho_2} - 1 \Rightarrow y + 1 = x$. So the only case that needs further investigation is when $y + 1 = x$. We consider two possible scenarios: ρ_1 satisfied the predicate with an (1) $\alpha < V+1$ and (2) $\alpha = V+1$.

Case 2(a): Here ρ_1 satisfied the predicate using an $\alpha < V+1$. This implies that $\cap_{m \in MS_{\rho_1}} m.seen$ might contain less than $V+1$ elements and thus not every virtual identifier will be included. So we have to consider two subcases: (1) $\nu_j \notin \cap_{m \in MS_{\rho_1}} m.seen$ and (2) $\nu_j \in \cap_{m \in MS1} m.seen$.

(1) Let first assume that $\nu_j \notin \cap_{m \in MS_{\rho_1}} m.seen$. Consider the set of servers $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$. Since $|MaxS_{\rho_1}| = |MS_{\rho_1}| \geq S - \alpha t$ and $|\mathcal{S}_{\rho_2}| = S - t$ then $|MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}| \geq S - (\alpha+1)t \geq 1$. Also since the read ρ_1 precedes ρ_2, and processes in $MaxS_{\rho_1}$ replied with $ts = TS_{\rho_1} = x$ to ρ_1, then processes in $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ reply with a timestamp $ts \geq x$ to ρ_2. So all the servers in the set $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ replied to ρ_2 with $ts = x = y + 1$. For any server $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$, let m_1 and m_2 be the messages of s_i in MS_{ρ_1} and MS_{ρ_2} respectively. We know that $m_1.ts = m_2.ts = x$. Since m_1 was sent before m_2, then $m_1.seen \subseteq m_2.seen$. Thus $\cap_{m \in MS_{\rho_1}} m.seen \subseteq \cap_{m \in MS_{\rho_2}} m.seen$. Moreover, every server $s_i \in MaxS_{\rho_2}$ adds ν_j into its $seen$ set before replying to ρ_2. Therefore clearly $\nu_j \in \cap_{m \in MS_{\rho_2}} m.seen$. By assumption though $\nu_j \notin \cap_{m \in MS_{\rho_1}} m.seen$, and so it follows that $|\cap_{m \in MS_{\rho_2}} m.seen| \geq |\cap_{m \in MS_{\rho_1}} m.seen| + 1 \geq \alpha + 1$. Since $|MaxS_{\rho_2}| = |MS_{\rho_2}|$ and $MaxS_{\rho_2} \geq |MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}| \geq S - (\alpha+1)t$, then the predicate for ρ_2 holds with $\alpha + 1$. Thus ρ_2 returns

$TS_{\rho_2} = x = y + 1$, which imposes a contradiction.

(2) Let now consider the case where $\nu_j \in \cap_{m \in MS_{\rho_1}} m.seen$. So either (i) a sibling of r_j or (ii) r_j itself performed a read operation before ρ_2. Assume that (i) r_j itself performed a read, say ρ_{2a}, before ρ_2. So since $\nu_j \in \cap_{m \in MS_{\rho_1}} m.seen$, r_j received a maximum timestamp $TS_{2a} = TS_{\rho_1}$ during read operation ρ_{2a}. In this case ρ_2 will represent a second read operation from r_j and so, during ρ_2, r_j sends a READ message with $ts = TS_{\rho_1} \geq x$. This message will be received by all servers in $\mathcal{S}_{\rho_2}$ which according to Lemma 4.3 will reply with a timestamp $ts' \geq TS_{\rho_1} \geq x$. If $ts' = x$ then the set of servers that replied with the maximum timestamp TS_{ρ_1} to ρ_2 will be $|MaxS_{\rho_2}| = |\mathcal{S}_{\rho_2}| \geq S - t$. Since every server $s_i \in \mathcal{S}_{\rho_2}$ before reply to ρ_2 adds ν_j to its $seen$ set, then predicate will hold for $\alpha = 1$. If now $ts' > x$, then ρ_2 will return a value y such that $y = ts'$ or $y = ts' - 1$ and thus in any case $y \geq x$. Both cases contradict with the assumption that $y + 1 = x$. Let now the case (ii) to be true and $\nu_j \in \cap_{m \in MS_{\rho_1}} m.seen$ because a sibling of r_j initiated a read operation before ρ_2. As we discussed above, $|MaxS_{\rho_2}| \geq |MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}| \geq S - (\alpha + 1)t$ and furthermore all the servers in $MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$ reply to ρ_2 with a timestamp $ts = x = y + 1$. Let m_1 and m_2 be the messages of a server $s_i \in MaxS_{\rho_1} \cap \mathcal{S}_{\rho_2}$, in MS_{ρ_1} and MS_{ρ_2} respectively. We know that $m_1.ts = m_2.ts$. Since m_1 is sent before m_2, then $m_1.seen \subseteq m_2.seen$. Thus $\cap_{m \in MS_{\rho_1}} m.seen \subseteq \cap_{m \in MS_{\rho_2}} m.seen$. Every server that replies to ρ_2, first adds ν_j into its $seen$ set and thus $\nu_j \in \cap_{m \in MS_{\rho_2}} m.seen$. Since though, ν_j was already in $\cap_{m \in MS_{\rho_1}} m.seen$, it follows that $|\cap_{m \in MS_{\rho_2}} m.seen| \geq |\cap_{m \in MS_{\rho_1}} m.seen| \geq \alpha$. If $|\cap_{m \in MS_{\rho_1}} m.seen| > \alpha$, then $|\cap_{m \in MS_{\rho_2}} m.seen| \geq \alpha + 1$. Since $|MaxS_{\rho_2}| = |MS_{\rho_2}| \geq S - (\alpha + 1)t$, the predicate holds with $\alpha + 1$ and ρ_2 returns $TS_{\rho_2} = x = y + 1$. If on the other hand $|\cap_{m \in MS_{\rho_1}} m.seen| = \alpha$ then ρ_1 will perform an informative operation before returning, sending the $postit = x$ to $|NS_{\rho_1}| \geq 2t + 1$ servers. So there will be a server $s_i \in \mathcal{S}_{\rho_2} \cap NS_{\rho_1}$ which will reply, by Lemma 4.4, with a $postit_{s_i} \geq x$ to ρ_2. So according to Lemma 4.7, ρ_2 will return a value $y \geq postit \geq x$. Hence we derive contradiction based on the initial assumption that $x = y + 1$.

Case 2(b): ρ_1 satisfied the predicate with an $\alpha = V + 1$. Since $|\{w, \nu_1, \ldots, \nu_V\}| = V + 1$ and $|\cap_{m \in MS_{\rho_1}} m.seen| \geq \alpha = V + 1$, it follows that $\nu_j \in \cap_{m \in MS_{\rho_1}} m.seen$. Observe that the set of servers that reply to ρ_1 with messages in MS_{ρ_1}, $|MaxS_{\rho_1}| \geq S - at > t$. So as shown in the previous case (Case 2(a)) ρ_2 will return a value $y \geq x$ deriving this way a contradiction. □

The main result of this section follows:

Theorem 4.11 *Algorithm SF implements an atomic read/write register in the SWMR model.*

Proof. Since the writer, any subset of readers and up to t servers might fail by crashing, we ensure termination in any execution of the implementation by letting any reader or writer to wait for messages only from $S - t$ servers during any communication round. Atomicity is preserved in any execution ξ by Lemmas 4.2, 4.8, 4.5, 4.10. Since both termination and atomicity properties are preserved the result follows. □

5 SF is a Semifast Implementation

In this section we show that the proposed implementation SF is a semifast implementation, that is, it satisfies all the properties of Definition 2.3. We use the same notation as in Section 3.

We first show that SF satisfies the third property of Definition 2.3.

Lemma 5.1 *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.*

Proof. Since ρ_2 might precede or succeed ρ_1 we examine the two cases separately. We proceed by considering an execution ξ of SF that contains both ρ_1 and ρ_2, and we show that in each case ρ_2 is fast or the case is not possible. For the rest of the proof we study the timestamps returned by the read operations since every value is associated with a unique timestamp. Let assume that timestamp $ts = k$ is associated with val_k, written by the unique write operation $\mathfrak{R}(\rho_1) = \mathfrak{R}(\rho_2) = \omega_k$.

Case 1: Starting from the case where $\rho_1 \rightarrow \rho_2$ there are two subcases to investigate: (1) ρ_2 observes a maximum timestamp equal to k, and (2) ρ_2 observes a maximum timestamp $k + 1$. Obviously in the second case, ρ_2 is concurrent with ω_{k+1} but ω_{k+1} is not yet completed.

The fast behavior of ρ_2 in the first subcase case follows from the fact that ρ_1 informs $|NS_{\rho_1}| \geq 2t + 1$ servers with the timestamp $ts = k$. So ρ_2 witnesses $|\mathcal{S}_{\rho_2} \cap NS_{\rho_1}| \geq t + 1$ postits equal to k during its first communication round. Since the maximum timestamp TS_{ρ_2} observed by ρ_2 is also equal to k, then ρ_2, according to Lemma 4.7, returns TS_{ρ_2} no matter the validity of the predicate. Moreover since $|\mathcal{S}_{\rho_2} \cap NS_{\rho_1}| \geq t + 1$ any subsequent read operation will witness at least one server in NS_{ρ_1} and thus ρ_2 completes without proceeding to a second communication round.

Consider now the second subcase where ρ_2 observes a maximum timestamp equal to $k + 1$. From the implementation we know that a read operation might return either the observed maximum timestamp TS_{ρ_2} or $TS_{\rho_2} - 1$. Since ρ_2 returned k, it implies that a decision for returning $TS_{\rho_2} - 1$ was taken by ρ_2. According though to the implementation, a reader may perform a second communication round only when it decides to return TS_{ρ_2}. In any other case the reader is not required to perform two communication rounds. So ρ_2 will return $TS_{\rho_2} - 1$ in one communication round as desired.

Case 2: Consider now the case where $\rho_2 \rightarrow \rho_1$. Since ρ_1 performs two communication rounds, it returns the maximum timestamp k, that ρ_1 observed during its first communication round. On the other hand ρ_2 also returns k, by either returning TS_{ρ_2} or $TS_{\rho_2} - 1$. So ρ_2 might observe a maximum timestamp $TS_{\rho_2} = k$ or $TS_{\rho_2} = k + 1$.

Lets first investigate the case where $TS_{\rho_2} = k + 1$. Recall that ρ_1 receives replies from $|\mathcal{S}_{\rho_1}| = S - t$ servers. Since ρ_1 observes a $TS_{\rho_1} = k$, then if $TS_{\rho_2} = k + 1$, it means that $k + 1$ was introduce to less than t servers

in the system. In order for ρ_2 to satisfy the predicate there must exist an MS_{ρ_2} which contains messages of the servers in $MaxS_{\rho_2}$, such that $|MS_{\rho_2}| \geq S - \alpha t$ for $\alpha \in \{1, \ldots, V+1\}$ and $V < \frac{S}{t} - 2$. Therefore we require that $|MS_{\rho_2}| \geq t$. However since $|MaxS_{\rho_2}| \geq |MS_{\rho_2}|$ and $|MaxS_2| \leq t$, we have that $|MS_{\rho_2}| \leq t$ and thus the predicate does not hold for ρ_2. Notice that for each read operation $\rho_i \rightarrow \rho_1$ (including ρ_2) and observing a maximum timestamp $TS_{\rho_i} = k+1$ the predicate is false and hence no read operation performed a second communication round informing the servers with a $postit = k+1$. So it follows that the second condition whether there are $postits = k+1$ will be false for ρ_2 as well and thus ρ_2 returns $TS_{\rho_2} - 1 = k$. As previously stated, if a read operation returns $TS - 1$ only needs one communication round.

It is left to examine now the case where ρ_2 observes a $TS_{\rho_2} = k$. Remember that ρ_1 performs a second communication round in two cases: (1) the predicate holds with $|\cap_{m \in MS_{\rho_1}} m.seen| = \alpha$ and (2) it observed "insufficient" postits sent by a concurrent read operation. For simplicity of our analysis we assume that no read operation was concurrent with ρ_1 and that ρ_1 performed a second communication round because the fist case was true. Since ρ_2 returns $TS_{\rho_2} = k$ then either (i) the predicate holds for ρ_2 or (ii) ρ_2 observed some $postit = k$. Let examine those subcases separately and we show that in each case ρ_2 is fast or the case is impossible.

Suppose that the predicate holds for ρ_2. So there is an $\alpha \in \{1, \ldots, V+1\}$ and there is $|MS_{\rho_2}| \geq S - \alpha t$ such that $|\cap_{m \in MS_{\rho_2}} m.seen| \geq \alpha$. If $|\cap_{m \in MS_{\rho_2}} m.seen| = \alpha$ then ρ_2 proceeds to a second communication round informing $|NS_{\rho_2}| \geq 2t+1$ servers about the maximum timestamp is about to return, $TS_{\rho_2} = k$. Since $\rho_2 \rightarrow \rho_1$, then $|\mathcal{S}_{\rho_1} \cap NS_{\rho_2}| \geq t+1$, and thus, ρ_1 would observe "enough" postists equal to$k = TS_{\rho_1}$ and would not perform a second communication round. This however contradicts our initial assumption, rendering this case impossible for ρ_2. Therefore the predicate validity is possible for ρ_2, only if $|\cap_{m \in MS_{\rho_2}} m.seen| > \alpha$. This is the case though where ρ_2 returns in one communication round as desired.

It remains to study the case where ρ_2 returns $TS_{\rho_2} = k$ because of some postits equal to k. There are two subcases to consider: (a) ρ_2 does not observe more than $t+1$ postits so it performs a second communication round and (b) ρ_2 observes more than $t+1$ postits and returns in one communication round. The first subcase will result in ρ_1 performing only one communication round as described above contradicting our initial assumption. In the second subcase there is a read operation ρ_i which is concurrent or precedes ρ_2 and performs two communication rounds. Since ρ_2 receives more than $t+1$ postits equal to TS_{ρ_2}, it returns in one communication round. Moreover, since we assumed that no read operation is concurrent with ρ_1, then ρ_i completes before the invocation of ρ_1. So ρ_i will inform at least $|NS_{\rho_i}| = 2t+1$ servers with a postit equal to k. Hence $|\mathcal{S}_{\rho_1} \cap NS_{\rho_i}| \geq t+1$ and thus ρ_1 returns in one communication round imposing contradiction. □

We now show that SF satisfies the fourth property of Definition 2.3. Notice that the proof of the lemma assumes *general* executions of our implementation SF which include both concurrent and non-concurrent operations. In fact the following proof assumes that all the read operations are concurrent with the write operation and yet are fast.

Lemma 5.2 *There exists an execution ξ of I that contains at least one write operation ω_k and the set of read operations $\mathcal{F} = \{\rho : \Re(\rho) = \omega_k\}$, such that $|\mathcal{F}| \geq 1$, $\exists \rho \in \mathcal{F}$, ρ is concurrent with ω_k and $\forall \rho \in \mathcal{F}$, ρ is fast.*

Proof. As in the previous proof we consider that each read operation $\rho \in \mathcal{F}$ returns the timestamp written by $\Re(\rho) = \omega_k$. Also assume that the timestamp written by ω_k is equal to $ts = k$. Recall that a read operation ρ returns either the maximum timestamp TS or $TS-1$. So the timestamp k is returned by ρ either when ρ witnesses $TS = k$ or when it witnesses $TS = k+1$. A read operation is fast in the following cases: (1) the predicate is correct and $|\cap_{m \in MS} m.seen > \alpha|$, or (2) more than $t+1$ postits equal to TS witnessed, or (3) the operation returns $TS-1$. In the contrary a read operation needs to perform two communication rounds when $|\cap_{m \in MS} m.seen = \alpha|$ or when ρ observed less than $t+1$ postits equal to TS in the replies from the servers.

Let assume, to derive contradiction, that for any execution ξ of I, that contains a write operation ω_k, $\exists \rho \in \mathcal{F}$ that returns $\Re(\rho) = \omega_k$ and is not fast. Consider the following finite execution fragment which is a prefix of ξ, $\varphi(wr)$. We assume that $\varphi(wr)$ contains the write operation ω_k performed by the writer w that writes timestamp k. Moreover, assume that $|\mathcal{S}_{\omega_k}| = S - \beta t$ servers received the WRITE messages from ω_k in $\varphi(wr)$, where $1 < \beta \leq V - 1$. Thus the write operation is incomplete.

We extent now $\varphi(wr)$ by the finite execution fragment $\varphi(1)$ which contains $\beta-2$ read operations $\rho_1, \ldots, \rho_{\beta-2}$ performed by $\beta - 2$ readers each of them from different virtual nodes. Let $\langle r_1, \nu_1 \rangle, \ldots, \langle r_{\beta-2}, \nu_{\beta-2} \rangle$ be the identifiers of the readers that invoked the read operations. Furthermore every reader $\langle r_i, \nu_i \rangle$ receive replies from all the servers that replied to the write operation. Hence each reader $\langle r_i, \nu_i \rangle$ witnesses an $|MS_{\rho_i}| = S - \beta t$ and an $|\cap_{m \in MS_{\rho_i}} m.seen| \leq \beta - 1$ and thus $|\cap_{m \in MS_{\rho_i}} m.seen| < \beta$. So the predicate condition is false for any read ρ_i from $\langle r_i, \nu_i \rangle$, returning timestamp $TS_{\rho_i} - 1$ in one communication round.

We further extent the execution fragment $\varphi(1)$ by execution fragment $\varphi(2)$ which contains two read operations performed by two sibling processes $\langle r_{1'}, \nu_{\beta-1} \rangle$ and $\langle r_{2'}, \nu_{\beta-1} \rangle$. Observe that those processes are not siblings with any of the previous readers. Let now the read operations $\rho_{1'}$ and $\rho_{2'}$ performed by the two sibling readers respectively, miss exactly t servers that received WRITE messages from ω_k. However, let them miss different t servers. For example, if the servers $s_1, \ldots, s_{2t}$ received WRITE messages, then $\rho 1'$ skips the servers $s_{t+1}, \ldots, s_{2t}$ and $\rho_{2'}$ skips the servers $s_1, \ldots, s_t$. Notice now that both readers will observe an $|\cap_{m \in MS_{\rho_{d'}}} m.seen| = \beta$, , for $d \in \{1, 2\}$, since they receive messages from servers that also replied to the read operations $\rho_1, \ldots, \rho_{\beta-2}$. However, both reads $\rho_{1'}$ and $\rho_{2'}$, since they miss t of the servers that received WRITE messages, they witness an $|MS_{\rho_{d'}}| = S-(\beta+1)t$. So the predicate is false for them as well and they return $TS_{\rho_{d'}} - 1$ in one communication round.

Finally we extend $\varphi(2)$ by $\varphi(3)$ which contains two read operations ρ_{1^*} by the reader $\langle r_{1^*}, \nu_\beta \rangle$ and ρ_{2^*} by the reader $\langle r_{2^*}, \nu_{\beta+1} \rangle$. Both readers are not siblings to any of the previous readers. We do not make any assumption about the relation of the two reads, that is, they may be concurrent. Let both reads receive messages from all

the servers that replied to the writer and thus $|MS_{\rho_{d^*}}| = S - \beta t$, again for $d \in \{1, 2\}$. Recall that any server $s_i \in MaxS_{\rho_{d^*}}$ contained a $seen = \{w, \nu_1, \ldots, \nu_{\beta-1}\}$, and before replying to ρ_{1^*} and ρ_{2^*}, they add ν_β and $\nu_{\beta+1}$ respectively in their $seen$ sets. Suppose that the intersection is $|\cap_{m \in MS_{\rho_{1^*}}} m.seen| = \beta + 1$ for ρ_{1^*} and $|\cap_{m \in MS_{\rho_{2^*}}} m.seen| = \beta + 2$ for ρ_{2^*}, that is, the servers replied to ρ_{1^*} before replying to ρ_{2^*}. Hence the predicate is correct by $|\cap_{m \in MS_{\rho_{d^*}}} m.seen| > \beta$ for both reads and thus they return $TS_{\rho_{d^*}} = k$ in one communication round. Notice that $\rho_{1^*}, \rho_{2^*} \in \mathcal{F}$ since $\mathfrak{R}(\rho_{1^*}) = \mathfrak{R}(\rho_{2^*}) = \omega_k$.

Any subsequent read operation ρ_ℓ by a reader $\langle r_j, \nu_g \rangle$ will witness an $|MS_{\rho_\ell}| \geq S - (\beta + 1)t$ and $|\cap_{m \in MS_{\rho_\ell}} m.seen| \geq \beta + 2$. So if ρ_ℓ witness $TS_{\rho_\ell} = k$ then the predicate will hold for ρ_ℓ and moreover will return $TS_{\rho_\ell} = k$ in one communication round. If ρ_ℓ returns k even though witnessed a maximum timestamp $TS_{\rho_\ell} = k + 1$ it would be also fast since any read operation that returns $TS_{\rho_\ell} - 1$ is fast. So by this construction we showed that there exists an execution ξ of I containing a write operation ω_k and all the read operations $\rho \in \mathcal{F}$ such that $\mathfrak{R}(\rho) = \omega_k$ are fast, contradicting our initial assumption. That completes our proof. □

We now state the main result of this section.

Theorem 5.3 *For any execution ξ, the proposed implementation SF is a semifast implementation of an atomic Read/Write Register.*

Proof. The properties (1) and (2) of Definition 2.3 are trivially satisfied since all the write operations as implemented by SF are fast and every read operation does not require more than two communication rounds to complete. Properties (3) and (4) of the same definition are ensured by lemmas 5.1 and 5.2. Thus our implementation SF is indeed a semifast implementation and that completes our proof. □

6 Impossibility of Obtaining Semifast Implementations

As it is shown in [3], no fast implementations exist if the number of readers R in the system is such that $R \geq \frac{S}{t} - 2$. Our approach to semifast solutions is to trade fast implementation for increased number of readers, while enabling some (many) reads to be fast. Here we show that semifast implementations are possible if and only if the number of virtual identifiers (virtual nodes) in the system is less than $\frac{S}{t} - 2$. We show that the bound on the virtual identifiers is tight for algorithms that: (1) consider each node acting individually in the system (as in [3]), and (2) consider weak grouping of the readers such that no reader is required to maintain knowledge of the membership of its own or any other group. (Related discussion appears in Section 2.) Throughout the section we assume the communication scheme presented and explained in Section 2.2.

Definitions and notation. We consider a system with V node groups (virtual nodes), such that $t \geq \frac{S}{(V+2)}$ (to derive contradiction). We partition the set of servers $\mathcal{S}$ into $V + 2$ subsets, called *blocks*, each denoted by B_i for

$1 \leq i \leq V+2$, where each block contains no more than t servers.

We say that an *incomplete operation* π *skips* a set of blocks BS in a finite execution fragment, where $BS \subseteq \{B_1, \ldots, B_{V+2}\}$, if : (1) no server in BS receives any READ or WRITE message from π, (2) all other servers receive messages and reply to π, and (3) those replies are in transit. A complete operation π that is fast is said to *skip* a block B_i in a finite execution fragment, where $B_i \in \{B_1, \ldots, B_{V+2}\}$ if: (1) no server in B_i receives a READ or WRITE messages from π, (2) all other servers receive the messages from π and reply, and (3) all replies are received by the process performing π. We say that an incomplete operation π that performs a second communication round *informs* a set of blocks BSI in a finite execution fragment, where $BSI \subseteq \{B_1, \ldots, B_{V+2}\}$ if: (1) all servers in BSI receive the INFORM message from π and reply, (2) those replies are in transit, and (3) no servers in any block $B_j \notin BSI$ receive any INFORM messages from π. A complete operation π that performs a second communication round *informs* a set of blocks BSI in an finite execution fragment, $BSI \subseteq \{B_1, \ldots, B_{V+2}\}$ if: (1) all servers in BSI receive the INFORM messages from π and reply, (2) no servers in any block $B_j \notin BSI$ receive any INFORM messages from π, and (3) those replies received by the process performing π. A complete operation π is said to be *skip-free* in an execution fragment if for every block B_i in the set $\{B_1, \ldots, B_{V+2}\}$, all the servers in B_i receive the messages from π and reply to them.

Block Diagrams. To facilitate the understanding of the proofs that follow, we provide schematic representations of block diagrams. We divide the diagram into columns each of them representing an operation (possibly incomplete) π, and at the bottom of each column we place an identifier of the invoking process in the form (r, ν), where r the actual id and ν the virtual id of the invoking process. Each column contains a set of rectangles. For an operation π if the i^{th} row of the column contains a rectangle it means that the servers in block B_i received a READ, INFORM or WRITE messages from π and replied to those messages. In other words we draw a rectangle in the i^{th} row of an operation π if π does not skip or informs the block B_i. If a rectangle is colored white, it means that block B_i received only a READ or WRITE messages from π. A two-color rectangle (black and white) in the i^{th} row of an operation π declares that the servers in block B_i received INFORM messages from π. If the operation identifier in a column is in a circle it means the operation is complete. Otherwise the operation has not yet completed. If the operation identifier is in a rectangle means that the operation is invoking the informative phase and has not yet received the required replies.

We now show that V cannot be greater or equal than $\frac{S}{t} - 2$. The idea behind the proof is to derive contradiction by assuming that semifast implementations exist for $V \geq \frac{S}{t} - 2$. We construct executions that violate atomicity and essential properties of the semifast definition. In particular we first assume an execution ξ which contains a skip-free write operation. We construct executions that can be extended to ξ, that contain fast read operations. We show that in execution extensions where the value of the write operation is propagated to less than t servers, some fast read operations return the value written, but others return an older value (since they may skip the servers with

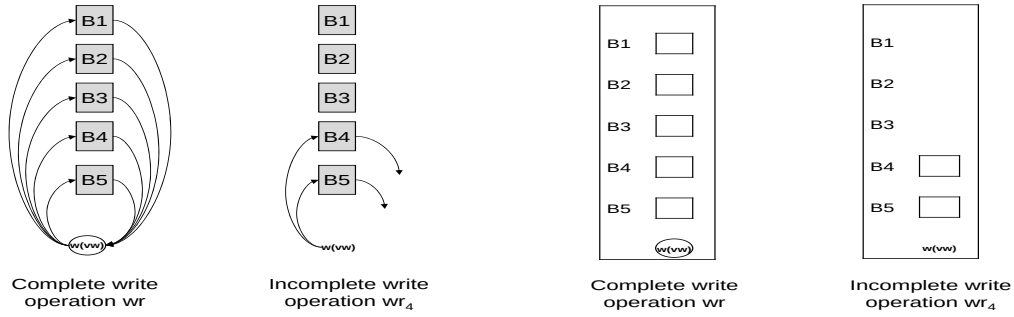


Figure 2: Left: Physical communication between w and the servers in $\varphi(wr)$ and $\varphi(wr_4)$. Right: Same communication using block diagrams.

the maximum timestamp). We emphasize that the first part of the proof can use the proof of Proposition 1 in [3] as a black box with the assumption of the skip-free write operation and the association of a distinct group id to each reader used. However we choose to present the proof here in its entirety for completeness. In the second part of the proof we present executions that violate atomicity even in the presence of a slow read operation.

Lemma 6.1 *No semifast implementation exists if the number of node groups V in the system is $\geq \frac{S}{t} - 2$.*

Proof. We proceed along the lines of Proposition 1 of [3]. We construct an execution of a semifast implementation I that violates atomicity. Namely we show that there exists an execution for I where some read returns 1 (the defined new value) and some subsequent read returns an older value, and in particular the initial value $\perp$. We consider two cases: (1) $V > \frac{S}{t} - 2$ and (2) $V = \frac{S}{t} - 2$. In the first case we show impossibility of the fast behavior if $V > \frac{S}{t} - 2$, thus violating property 4 of Definition 2.3. In case (2) we show that there exists an execution where atomicity is violated even in the presence of a two-round read operation. This violates property 3 of Definition 2.3.
Case 1: Since $V > \frac{S}{t} - 2$, it suffices to show that we derive contradiction in the case where $V \geq \frac{S}{t} - 1$. So we can partition the set of servers into $V + 1$ blocks $\{B_1, \ldots, B_{V+1}\}$ where each block contains $\leq t$ servers. We provide the constructions we use for the needs of this proof in the write and read operation paragraphs and then we present an execution scenario based on those constructions that violates atomicity.
Write Operations. Let $\varphi(wr)$ be an execution fragment in which operation ω_1 is completed by w. Let the operation be *skip-free*; this is the best case for a write operation and thus our lower bound applies to all other possible cases. We define a series of finite execution fragments which can be extended to $\varphi(wr)$. We say that in the finite execution fragment $\varphi(wr_{V+2})$ the writer w has invoked a ω_1 operation, but all the WRITE messages are in transit. Then, for $1 \leq i \leq V+1$, we say that $\varphi(wr_i)$ is the finite execution fragment that contains an incomplete write(1) operation and skips the set of blocks $\{B_j | 1 \leq j \leq i-1\}$. Observe that: (1) the finite execution fragments $\varphi(wr_i)$ and $\varphi(wr_{i+1})$ differ only on block B_i, (2) since in $\varphi(wr_1)$ we do not skip any block but all the replies are in transit, then $\varphi(wr)$ is an extension of $\varphi(wr_1)$ where all those replies are received by w and (3) only w can distinguish $\varphi(wr)$ from $\varphi(wr_1)$. Figure 2 illustrates the communication between the writer w and the groups of

servers in the finite execution fragments $\varphi(wr)$ and $\varphi(wr_3)$. The figure shows both physical communication and block diagram representation.

Read Operations. We now construct finite execution fragments for read operations. We assume that only one reader process, say r_j, owns a virtual identifier ν_j and denoted by the pair $\langle r_j, \nu_j \rangle$. Let $\varphi(1)$ be a finite execution fragment that extends $\varphi(wr)$ by containing a complete read operation by a reader with id $\langle r_1, \nu_1 \rangle$ that skips B_1. Consider now $\varphi'(1)$ to be an extension of $\varphi(wr_2)$ that appends $\varphi(wr_2)$ by a complete read operation by the reader $\langle r_1, \nu_1 \rangle$ that skips B_1. Notice that reader $\langle r_1, \nu_1 \rangle$ cannot distinguish $\varphi(1)$ from $\varphi'(1)$ because $\varphi(wr)$ and $\varphi(wr_2)$ differ at w and block B_1 and read from $\langle r_1, \nu_1 \rangle$ skips block B_1.

We continue in similar manner, starting from $\varphi'(1)$, and create execution fragments for the rest of the readers in the system. In particular we define an execution fragment $\varphi(i)$, for $2 \leq i \leq V$ to extend $\varphi'(i-1)$ by a complete read operation from $\langle r_i, \nu_i \rangle$ that skips B_i. We then construct finite execution fragment $\varphi'(i)$ by deleting from $\varphi(i)$ all the rectangles (steps) from the servers in block B_i. In particular, as previously mentioned, execution fragment $\varphi'(i)$ extends $\varphi(wr_{i+1})$ by appending that with i reads such that for $1 \leq k \leq i$, $\langle r_k, \nu_k \rangle$ skips the blocks $\{B_j |\ k \leq j \leq i\}$. Observe that since $\langle r_1, \nu_1 \rangle$ cannot distinguish $\varphi(1)$ and $\varphi'(1)$, it returns 1 in both executions. Furthermore, since $\varphi(2)$ extends $\varphi'(1)$, by atomicity $\langle r_2, \nu_2 \rangle$ returns 1. So $\langle r_2, \nu_2 \rangle$ returns 1 in $\varphi'(2)$ since it cannot distinguish $\varphi(2)$ and $\varphi'(2)$. By following inductive arguments we conclude that for $\varphi'(i)$, reader $\langle r_i, \nu_i \rangle$ returns 1. Thus, for the execution fragment $\varphi'(V)$, $\langle r_V, \nu_V \rangle$ returns 1. An illustration of the following execution fragments can be seen in Figure 3.

Finite Execution fragment $\varphi(A)$. Here we consider the execution fragment $\varphi'(V)$. As defined above, $\varphi'(V)$ extends $\varphi(wr_{V+1})$ by appending V reads such that for $1 \leq k \leq V$, $\langle r_k, \nu_k \rangle$'s read skips the blocks $\{B_j |\ k \leq j \leq V\}$. Observe here that all the read operations are incomplete except for the read operation of reader $\langle r_V, \nu_V \rangle$. Moreover only the servers in block B_{V+1} receive WRITE messages from the ω_1 operation of w. Also, only B_{V+1} replies to the read operation of the reader $\langle r_1, \nu_1 \rangle$, and those messages are in transit. All other READ messages of $\langle r_1, \nu_1 \rangle$ are in transit and not yet received by any other server. Let execution fragment $\varphi(A)$ extend $\varphi'(V)$ as follows: (1) all the messages send by $\langle r_1, \nu_1 \rangle$ and were in transit, are received by the servers in blocks $B_1, \ldots, B_V$, (2) reader $\langle r_1, \nu_1 \rangle$ receives the replies from servers $B_1, \ldots, B_V$, and returns from the read operation. Notice that since B_{V+1} contains less or equal to t servers, it means that reader $\langle r_1, \nu_1 \rangle$ received $\geq S - t$ replies and should not wait for any more replies to return.

Finite Execution fragment $\varphi(B)$. We consider as execution fragment $\varphi(B)$ with the same communication pattern as $\varphi(A)$ but with the difference that the ω_1 operation is not invoked at all. Hence servers in block B_{V+1} do not receive any WRITE messages. Clearly only the servers in block B_{V+1}, the writer and the readers $\langle r_2, \nu_2 \rangle$ to $\langle r_V, \nu_V \rangle$ are in position to distinguish $\varphi(A)$ from $\varphi(B)$. The reader $\langle r_1, \nu_1 \rangle$, since it does not receive any messages from B_{V+1} cannot distinguish $\varphi(A)$ from $\varphi(B)$. So, since there is no write (ω_*) operation, $\langle r_1, \nu_1 \rangle$ returns $\perp$ in

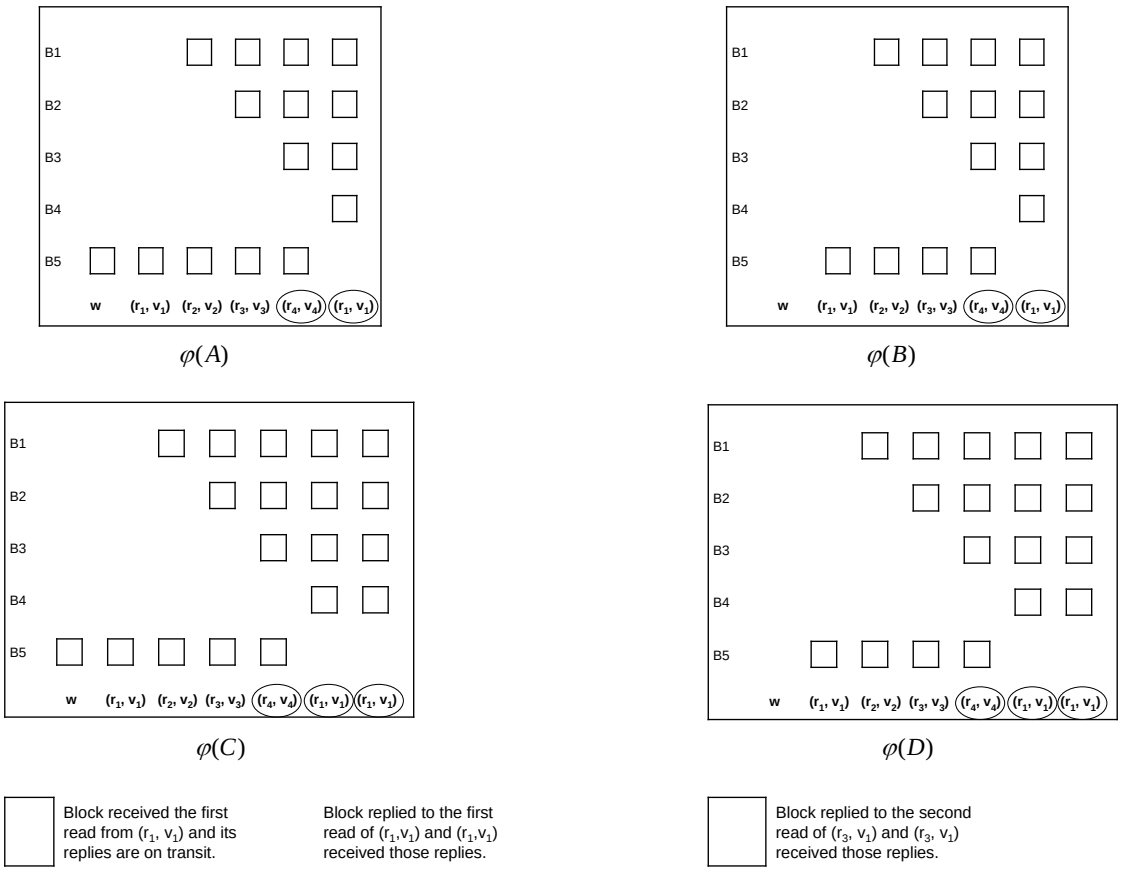


Figure 3: Execution fragments $\varphi(A)$, $\varphi(B)$, $\varphi(C)$, $\varphi(D)$.

$\varphi(B)$ and therefore returns $\perp$ in $\varphi(A)$ as well.

Finite Execution fragments $\varphi(C)$ and $\varphi(D)$. Observe that in $\varphi(A)$, reader $\langle r_1, \nu_1 \rangle$ does not violate atomicity even though it returns $\perp$ and $\langle r_V, \nu_V \rangle$ returns 1 because the two operations are concurrent. We construct now two more executions: execution fragment $\varphi(C)$ and $\varphi(D)$ which extend the execution fragments $\varphi(A)$ and $\varphi(B)$ respectively with a second complete read operation from $\langle r_1, \nu_1 \rangle$ that skips B_{V+1}. Since the servers in B_{V+1} are the only ones who can distinguish $\varphi(A)$ and $\varphi(B)$ and since $\langle r_1, \nu_1 \rangle$'s second read skips B_{V+1} then $\langle r_1, \nu_1 \rangle$ cannot distinguish $\varphi(C)$ from $\varphi(D)$ either. Since $\varphi(C)$ is an extension of $\varphi(A)$ means that the reader $\langle r_V, \nu_V \rangle$ returns 1 in $\varphi(C)$. Moreover $\langle r_1, \nu_1 \rangle$ returns $\perp$ since no write (ω_*) operation is invoked in $\varphi(D)$. So since $\langle r_1, \nu_1 \rangle$ cannot distinguish $\varphi(C)$ from $\varphi(D)$, it returns $\perp$ in $\varphi(C)$ as well. However $\langle r_1, \nu_1 \rangle$ succeeds $\langle r_V, \nu_V \rangle$ that returns 1 in $\varphi(C)$ and thus *violates atomicity*. This completes the proof of Case (1).

Case 2: The next case that needs investigation is the equality $V = \frac{S}{t} - 2$. Since we are using groups of nodes, it is possible that all the readers will be contained in a single group. Consider this situation for the following proof. As before, since $V = \frac{S}{t} - 2$ we can divide the servers into $V + 2$ blocks where each block contains t servers. More precisely, since we only assume one virtual node ($V = 1$) then the total number of blocks is 3. We also consider the same construction for the write operation with the difference that the ω_1 is not skip-free but skips the block B_{V+2}.

In particular $\varphi(wr_i)$ is the execution fragment that contains an incomplete ω_1 operation and skips the set of blocks $\{B_{V+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. As before, $\varphi(wr_1)$ is the execution where all the servers $\{B_j |\ 1 \leq j \leq V+1\}$ replied to the ω_1 write operation and all those replies are in transit. So $\varphi(wr)$ is the extension of $\varphi(wr_1)$ where all those replies are being received by the writer w.

Let now describe a series of finite execution fragments that extend $\varphi(wr)$. We say that execution fragment $\varphi(e1)$ extends $\varphi(wr)$ by a complete read operation by the reader $\langle r_1, \nu_1 \rangle$ that skips block B_1. To preserve atomicity, $\langle r_1, \nu_1 \rangle$ returns 1. Consider now another execution, $\varphi'(e1)$, that extends $\varphi(wr_2)$ by the same read operation from $\langle r_1, \nu_1 \rangle$ that again skips B_1. Recall that only the writer w and the servers in block B_1 can distinguish $\varphi(wr)$ from $\varphi(wr_2)$. So since $\langle r_1, \nu_1 \rangle$ skips the servers in the block B_1, it cannot distinguish $\varphi(e1)$ from $\varphi'(e1)$ and thus returns 1 in $\varphi'(e1)$ as well. We now extend $\varphi'(e1)$ by execution fragment $\varphi(e2)$ as follows: (1) a complete inform(1) operation from $\langle r_1, \nu_1 \rangle$ that skips the servers in the block B_{V+2}, and (2) a complete read operation from reader $\langle r_2, \nu_1 \rangle$ that skips block B_1. The read from $\langle r_2, \nu_1 \rangle$ returns 1 to preserve atomicity. Further consider the execution fragment $\varphi(e3)$ which is the same with $\varphi(e2)$, but with the difference that the inform operation from $\langle r_1, \nu_1 \rangle$ is incomplete and also skips block B_1. Notice that $\varphi(e2)$ and $\varphi(e3)$ differ at the reader $\langle r_1, \nu_1 \rangle$ and the servers in block B_1 only. Since the reader $\langle r_2, \nu_1 \rangle$ does not receive any messages from B_1, it cannot distinguish the two executions. Therefore $\langle r_2, \nu_1 \rangle$ returns 1 in $\varphi(e3)$ as well.

It now remains to investigate two more execution fragments, $\varphi(E)$ and $\varphi(F)$. Let $\varphi(E)$ extend $\varphi(e3)$ with a complete read operation by $\langle r_3, \nu_1 \rangle$. This read operation skips block B_2. The read from $\langle r_2, \nu_1 \rangle$ cannot distinguish $\varphi(E)$ from $\varphi(e3)$ and so it returns 1 in $\varphi(E)$ as well. Execution $\varphi(F)$ now has the same configuration as $\varphi(E)$ with the difference that no write (ω_*) or inform(*) operation is invoked by any process. So $\langle r_1, \nu_1 \rangle$, $\langle r_2, \nu_1 \rangle$ and $\langle r_3, \nu_1 \rangle$ return $\perp$ in $\varphi(F)$. However, since $\varphi(E)$ and $\varphi(F)$ only differ at block B_2, and since $\langle r_3, \nu_1 \rangle$ skips B_2, it cannot differentiate the two executions fragments. Hence, $\langle r_3, \nu_1 \rangle$ returns $\perp$ in $\varphi(E)$ as well. Therefore, $\varphi(E)$ *violates atomicity* since $\langle r_2, \nu_1 \rangle$ that succeeds $\langle r_3, \nu_1 \rangle$ returns 1 and $\langle r_3, \nu_1 \rangle$ returns an older value, namely $\perp$. This completes the proof. □

Per Lemma 6.1 semifast implementation are possible only if $V < \frac{S}{t} - 2$. In addition, the following lemma shows that the existence of a semifast implementation also depends on the number of minimum messages sent by a process during its second communication round.

Lemma 6.2 *There is no semifast implementation of an atomic register if a read operation informs $3t$ or fewer servers during its second communication round.*

Proof. Since $V < \frac{S}{t} - 2$, we get that $S > t(V+2)$, and hence in order to maintain at least one reader in the system, $S > 3t$. Suppose by contradiction that there exist a semifast implementation I which requires a complete read operation to send equal to $3t$ INFORM messages during its second communication round. Recall that the reader that

performs the informative phase, in order to preserve the termination property, should expect $2t$ replies (since up to t servers might fail). We proceed by showing that there exists an execution of I where a read operation returns 1 and performs a second communication round and a subsequent read operation returns 1 and again needs to perform a second communication round to complete, *violating* the third property of the semifast implementation.

Consider a finite execution fragment φ_1 where writer w invokes a ω_k write operation and writes the value val_k on the atomic register. We extend φ_1 by a read operation ρ_1 which performs two communication rounds and returns val_k. During the second communication round, ρ_1 sent messages to $3t$ servers. Only $|NS_{\rho_1}| = 2t$ servers get INFORM messages from ρ_1 and replied to those messages. Since t of the servers might be faulty, in order to preserve the termination property, ρ_1 returns after the receipt of those replies. We further extend φ_1 by a second read operation ρ_2, which receives messages from $|\mathcal{S}_{\rho_2}| = S - t$ servers and misses t of the servers in NS_{ρ_1} such that $|\mathcal{S}_{\rho_2} \cap NS_{\rho_1}| = t$.

We now describe a second finite execution fragment φ_2 which is similar to φ_1 but with the difference that ρ_1 is incomplete and only $|NS_{\rho_1}| = t$ servers received the INFORM messages from ρ_1. In this execution, ρ_2 receives replies from all the servers that have been informed by ρ_1, namely $|\mathcal{S}_{\rho_2} \cap NS_{\rho_1}| = t$. Note that ρ_2 cannot distinguish φ_1 and φ_2 in terms of the number of servers informed by ρ_1. Since ρ_2 observed that only t servers were informed by ρ_1 in φ_2 and since ρ_1 might crash before completing, ρ_2 must perform a second communication round to ensure that any read operation s.t. $\rho_2 \rightarrow \rho_i$ that receives replies from $|\mathcal{S}_{\rho_i}| = S - t$ servers will not observe $|\mathcal{S}_{\rho_i} \cap NS_{\rho_1}| = 0$ and thus return an older value violating atomicity. Obviously the fact that ρ_2 proceeds to a second communication round does not violate the third property of Definition 2.3 since ρ_1 and ρ_2 in φ_2 are concurrent. Since ρ_2 cannot distinguish φ_1 and φ_2, ρ_2 must perform a second communication round in φ_1 as well. However, in φ_1, $\rho_1 \rightarrow \rho_2$ and thus they are not concurrent. So φ_1 *violates the third property*, contradicting the assumption that there is a semifast implementation I, where any read operation needs to inform $\leq 3t$ servers. □

We now state the main result of this section.

Theorem 6.3 *No semifast implementation I exists if the number of virtual nodes in the system is $\geq \frac{S}{t} - 2$ and if $3t$ or fewer servers are informed during a second communication round.*

Proof. It follows directly from Lemmas 6.1 and 6.2. □

7 Multiple Writer, Multiple Reader (MWMR) Model

In this section we consider the MWMR model and show that no semifast implementations of atomic registers are possible in this setting in the presence of server failures.

7.1 Preliminaries.

For the MWMR model we relax the definition of a semifast implementation as presented for the SWMR model, by allowing read operations to perform more than two communication rounds (i.e., instead of two rounds we allow multiple rounds in Definition 2.3). First we extract several immediate properties from the definition of atomicity presented in Section 2. To satisfy the atomicity definition the following properties must be true for any execution of the MWMR semifast implementation:

> PROPERTY P1: if there is a write operation ω_k that writes value val_k and a read operation ρ_i such that $\omega_k \rightarrow \rho_i$, and all other writes precede ω_k then ρ_i returns val_k.
>
> PROPERTY P2: if the response steps of all write operations precede the invocation steps of the read operations ρ_i and ρ_j, $i \neq j$, then ρ_i and ρ_j must return the same value.
>
> PROPERTY P3: If the response steps of all the write operations precede the invocation step of a read operation ρ_i then ρ_i returns a value written by some complete write.

For the reasons discussed in Section 2.2, we assume the communication scheme where a server replies to a READ (or WRITE or INFORM) message without waiting to receive any other READ (or WRITE or INFORM) messages. In this proof we say that an operation performs a *read phase* during a communication round if it gathers information regarding the value of the object from the system at that round. We say that an operation performs a *write phase* during a communication round if it propagates information regarding the value of the object to any subset of the servers at that round. A read phase of an operation (read or write) does not modify the value of the atomic object. On the other hand a write phase of an operation π behaves as follows according to its type: (1) a new, currently unknown value is written to the register, if π is a write operation (2) only previously known values are written to the register if π is a read operation. We should clarify here that by *"value of the atomic object"* we mean all the parameters that may reveal any information about the actual value of the object. So any operation phase that modifies those parameters (and thus the state of the atomic object) is considered as a write phase.

We say that a complete operation π *skips* a server s_i if s_i does not receive any messages from the process p that invoked π and the process p does not receive any replies from s_i. All other servers that receive the READ, WRITE or INFORM messages from p reply to these, and p receives those replies. All other messages remain in transit. Since we assume that $t = 1$, any complete operation may skip at most one server. We say that an operation is *skip-free* if it does not skip any server.

Since we consider read operations that might perform multiple communication rounds to complete, we denote by $\rho_i(j)$ the j^{th} communication round (phase) of a read operation ρ_i. In order to distinguish between the read and write phases of ρ_i, let $\rho_i^{\omega}(j)$ denotes that the j^{th} phase of the read ρ_i is a write phase. An arbitrary delay may occur between two phases $\rho_i(j)$ and $\rho_i(j+1)$ where other read (write) operations or read (write) phases might

be executed. So we define as $sr_i(j-1)$ a set of operation phases (read or write) with the property that any phase $\rho_*(*) \in sr_i(j-1)$, $\rho_*(*) \rightarrow \rho_i(j)$. A set $sr_i(j-1)$ might be equal to the empty set containing no operations.

Claim 7.1 *A read operation ρ that succeeds any write operation ω_* and write phase $\rho_*^{\omega}(*)$ from an operation $\pi \neq \rho$, returns the value decided by the read phase that precedes its last write phase.*

Proof. The claim follows from the fact that the read operation succeeds all the write operations and from atomicity properties P1 and P2. Let assume that reader r_i performs the read operation ρ which in turn requires n communication rounds to complete. Furthermore let assume that $\rho^{\omega}(j)$ is the last *write phase* of ρ and for simplicity of analysis we also assume that this is the only write phase from ρ. The result is still valid when multiple write phases are performed by ρ.

Since ρ succeeds all write operations then any read phase $\rho(g)$, for $1 \leq g \leq n$ where n the total number of phases from ρ, will gather the same information about the value of the atomic register. So according to r_i's local policy and atomicity property P3 every read phase that precedes $\rho^{\omega}(j)$ will decide the same value, say v to be the latest value written on the register. Let $\rho(j-1)$ be the last read phase operation that precedes $\rho^{\omega}(j)$. According to the assumption, a write phase of a read operation propagates the value gathered, to the system. So $\rho_i^{\omega}(j)$ propagates value v which was observed by the read phases. Since $\rho^{\omega}(j)$ performs a write operation on the register then any read phase $\rho(\ell)$, $j+1 \leq \ell \leq n$, such that $\rho^{\omega}(j) \rightarrow \rho(\ell)$ must decide v to preserve atomicity property P1. So the last read phase $\rho(n)$ of the read operation returns v as well and hence v is the value returned by operation ρ. That completes the proof. □

7.2 Construction and Main Result.

We now present the construction we use to prove the main result. We show execution constructions assuming that two writers (w_1 and w_2), and two readers (r_1 and r_2) participate in the system. We assume skip-free operations since they comprise the best case scenario and thus a lower bound for these is sufficient. Note here that the constructions of executions with fast read operations are similar to constructions presented in [3]. We use this approach and we present a generalization that contains read operations with single or multiple communication rounds suitable for our exposition. The main idea of the proof exploits executions with certain ordering assumptions which may violate atomicity. In particular we assume executions where the two write operations are concurrent and interleaved, are succeeded by the read ρ_1, and in turn ρ_1 is succeeded by ρ_2. We analyze all the different cases in terms of communication rounds for ρ_1 and ρ_2. We show that in each case, a single server failure may cause violations of atomicity.

Let us first consider the finite execution fragment φ_1, constructed from the following skip-free, complete operations: (a) operation $write(2)$ by w_2, (b) operation $write(1)$ by w_1, and (c) operation ρ_1 by r_1. These

operations are not concurrent and they are executed in the order $write(2) \rightarrow write(1) \rightarrow \rho_1$. By property P2, operation ρ_1 returns 1.

We now invert the write operations of the above execution and we obtain execution φ_2, consisting of the following skip-free, complete operations in the following order: (a) operation $write(1)$ by w_1, (b) operation $write(2)$ by w_2, and (c) operation ρ_1 by r_1. As before, these operations are not concurrent. So in this case, by property P2, operation ρ_1 returns 2.

The generalization φ_{1g} of φ_1, for $1 \leq i \leq n$, when the reader r_1 performs n communication rounds is the following: (a) a $write(2)$ operation from w_2, (b) a $write(1)$ operation from w_1, (c) a set of read operations $sr_1(i-1)$ from readers r_j, $j \neq 1$, and (d) a read or a write phase $\rho_1(i)$ of the ρ_1 operation from reader r_1. Notice that for $n = 1$ and for $sr_1(0) = \emptyset$ no process can distinguish φ_{1g} from φ_1. Clearly at the end of phase $\rho_1(n)$, by property P2, the operation ρ_1 from r_1 returns 1.

Similarly we define the φ_{2g} to be the generalization of φ_2, where the write operations are inversed: (a) a $write(1)$ operation from w_1, (b) a $write(2)$ operation from w_2, (c) a set of read operations $sr_1(i-1)$ from readers r_j, $j \neq 1$, and (d) a read or a write phase $\rho_1(i)$ of the ρ_1 operation from reader r_1. In this case by the end of phase $\rho_1(n)$, and by property P2, the ρ_1 operation returns 2.

If we assume now, without loss of generality, that the last communication round $\rho_1(n)$ of r_1 in φ_{1g} is a write phase, thus $\rho_1^{\omega}(n)$, then r_1 should not be able to differentiate φ_{1g} from the following execution, for $1 \leq i \leq n-1$: (a) a $write(2)$ operation from w_2, (b) a $write(1)$ operation from w_1, (c) a set of read operations $sr_1(i-1)$ from readers r_j, $j \neq 1$, (d) a read phase $\rho_1(i)$ of the ρ_1 operation from reader r_1, (e) a set of read operations $sr_1(n-1)$ from readers r_j, $j \neq 1$, and (f) a $write(1)$ operation from $\rho_1^{\omega}(n)$. By operation $write(1)$, the reader r_1 tries to disseminate the information gathered from the previous rounds regarding the value of the atomic object. Similarly we can define φ_{2g} with the difference that reader r_1 will perform a $write(2)$ operations during its last communication round.

Obviously we have the same setting as in Claim 7.1 and so by the same claim the decision for the return value must be made in $\rho_1(n-1)$. Notice that the decision of r_1 taken in $\rho_1(n-1)$ is not affected from the operations in $sr_1(n-1)$. So we can assume that φ_{1g} and φ_{2g} contain only read phases by r_1. According now to property P2, r_1 will decide 1 by the end of $\rho_1(n-1)$ in φ_{1g} and 2 by the end of $\rho_1(n-1)$ in φ_{2g}. Since we assume that we only have 2 readers in the system r_1 and r_2, and since r_2 does not perform any read operation in either φ_{1g} or φ_{2g}, we have that all the sets $sr_1(i-1) = \emptyset$ for $1 \leq i \leq n$ in both executions φ_{1g} and φ_{2g}.

Theorem 7.2 *If the number of writers in the system is $W \geq 2$, the number of readers is $R \geq 2$, and $t \geq 1$ servers may fail, then there is no semifast atomic register implementation.*

Proof. It suffices to show that the theorem holds for the basic case where $W = 2$, $R = 2$, and $t = 1$. We assume that there exists a semifast implementation and we derive a contradiction. Let w_1 and w_2 be the writers, r_1 and r_2

the readers, and $s_1, \ldots, s_S$ the servers participating in the system. We show a series of executions and analyze the different cases of a semifast implementation where writers are fast and readers perform n communication rounds. We show that in all of these cases atomicity can be violated.

We now define a series of finite execution fragments $\varphi(i)$, where $1 \leq i \leq S+1$. We assume that the two write operations from w_1 and w_2 are concurrent. After the completion of both write operations a ρ_1 read operation, which may involve multiple communication rounds (phases), is invoked by r_1. For every $\varphi(i)$ the set of read operations $sr_1(0) = \emptyset$ and so the ρ_1 from r_1 is the first read after the completion of the write operations. Define $\varphi(1)$ to be similar to φ_{1g}. Then we iteratively define $\varphi(i+1)$ to be similar to $\varphi(i)$ except that server s_i receives the message from w_1 before the message from w_2. In other words the arrival order of the write messages are interchanged in s_i. Since the operations from w_1, w_2 and each communication round by r_1 are skip-free, they can differentiate between $\varphi(i)$ and $\varphi(i+1)$. Also s_i is the only server that can distinguish the two executions since we assume no communication between the servers. Obviously, by our construction, no server can distinguish $\varphi(S+1)$ from φ_{2g} since every server received the WRITE messages in the opposite order than in φ_{1g}. Thus, r_1 cannot distinguish the two executions either, and so it returns 2 in $\varphi(S+1)$ after the completion of its last communication round. Therefore, executions $\varphi(S+1)$ and φ_{2g} differ only at w_1 and w_2. It follows that since r_1 returns 1 in $\varphi(1)$, 2 in $\varphi(S+1)$ and 1 or 2 in $\varphi(i)$ ($2 \leq i \leq S$), there are two executions $\varphi(m)$ and $\varphi(m+1)$, such that $1 \leq m \leq S$ and the read by r_1 returns 1 in $\varphi(m)$ and 2 in $\varphi(m+1)$ at the end of the same communication round.

Consider now an execution fragment φ' and an execution fragment φ'' that extend $\varphi(m)$ and $\varphi(m+1)$ respectively by a read operation ρ_2 from r_2 that skips s_m during all its required communication rounds. On the constructed executions we analyze the cases of semifast implementation. Recall that we investigate the case of the semifast implementation where we allow the readers to perform n communication rounds and write operations are fast (only one communication round). We examine the different possible scenarios during executions φ' and φ'': (1) both ρ_1 and ρ_2 are fast in both executions, (2) ρ_2 performs k communication rounds in φ' and φ'' and ρ_1 is fast, (3) ρ_1 performs n communication rounds in both executions and ρ_2 is fast, and (4) both ρ_1 and ρ_2 perform n and k communication rounds respectively. We assume that the processes decide to perform a second communication round according to their local policy.

Case 1: In this case both reads are fast and thus requiring only one communication round to complete. As shown in Proposition 2 in [3] the read operation ρ_2 cannot distinguish the two executions φ' and φ'' since it skips the only server (s_m) that can differentiate them. So the read ρ_2 returns, according to property P2, 1 in φ' and so it returns 1 in φ'' as well. However, ρ_1 cannot distinguish the executions $\varphi(m+1)$ and φ'', and so, since it returns 2 in $\varphi(m+1)$, it returns 2 in φ'' as well. Hence, φ'' violates property P2.

Case 2: In this case ρ_2 performs k phases in executions φ' and φ''. Since all read phases by ρ_2 skip the server s_m, then none of them is able to distinguish execution φ' from φ'' since s_m is the only server who can differentiate

them. Thus ρ_2 retuns the same value in both executions. Since according again to P2 ρ_2 returns 1 in φ' then it returns 1 in φ'' as well. Again ρ_1 cannot distinguish $\varphi(m+1)$ from φ'' so it returns 2 in φ'' as well. Thus property P2 is again violated.

Case 3: This is the case where ρ_1 performs n phases to complete and ρ_2 is fast. Since all the phases by ρ_1 are read phases, skip-free and precede ρ_2, then ρ_1 cannot distinguish execution φ' from $\varphi(m)$ and φ'' from $\varphi(m+1)$. Therefore ρ_1 returns 1 in φ' and 2 in φ''. On the other hand, ρ_2 returns according to property P2 1 during φ'. Since all n phases of r_1 are read phases in both executions φ' and φ'' , then no server, writer or r_2 can distinguish each phase and they only differ at r_1. So only s_m differentiates φ' from φ''. Since though ρ_2 skips s_m then it cannot distinguish φ' from φ'' returning 1 in φ'' as well violating property P2.

Case 4: Similarly to case 3, ρ_1 returns 1 during φ' and 2 during φ''. With the same reasoning as in case 3 and since all phases of ρ_2 skip the server s_m, no communication round of ρ_2 can distinguish φ' from φ''. So in this case ρ_2 returns 1 in both executions violating property P2. This completes the proof. □

8 Simulation Results

To evaluate our implementation, we simulated algorithm SF using the NS-2 network simulator and measured the percentage of two-round read operations as a function of the number of readers and the number of faulty servers.

Our simulations include 20 servers ($S = 20$). To guarantee liveness we need to constrain the maximum number of server failures t so that $V < \frac{S}{t} - 2$ or $V \leq \frac{S}{t} - 3$. Thus $t \leq \frac{S}{V+3}$. In order to maintain at least one group ($V = 1$), t must not exceed $\frac{S}{4}$, or 5 failures. Thus in our simulations we allow up to 5 servers to fail at arbitrary times. We vary the number of reader processes between 10 and 80.

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.

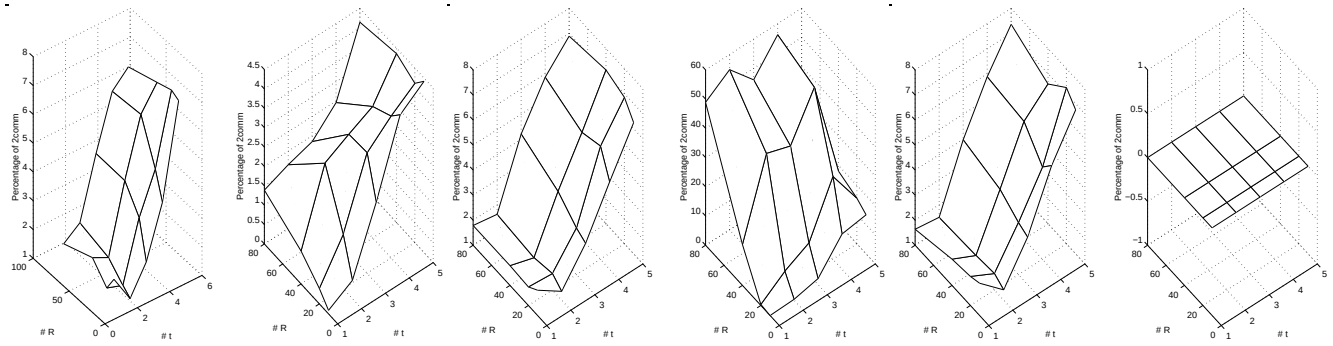
We considered three simulation scenarios corresponding to the following parameters:

(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.

In our simulation setting the nodes are connected with duplex links at 1MB bandwidth, a latency of $10ms$, and a DropTail queue. To model asynchrony, the processes send messages after a random delay between 0 and 0.3 sec (smaller than $rInt$ and $wInt$). 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).



4.a(i) 4.b(i) 4.a(ii) 4.b(ii) 4.a(iii) 4.b(iii)

Setting a: Stochastic simulations Setting b: Fixed interval simulations

Figure 4: Stochastic and fixed interval simulations. The vertical axes show the percentage of two-round reads as a function of the number of readers and the number of faulty servers.

Each of the simulation scenarios (i), (ii), and (iii) was considered in two settings:

a. Stochastic simulations, where the intervals between two read or two write operations vary randomly within certain bounds.

b. Fixed interval simulations, where the intervals are fixed.

We now describe the simulation results for each of the two settings.

Setting a: Stochastic simulations. Here we consider a class of executions where each read (resp. write) operation from an invoking process is scheduled at random time between 1 sec and $rInt$ (resp. $wInt$) after the last read (resp. write) operation. Introducing randomness in the operation invocation intervals renders a more realistic scenario where processes are interacting with the atomic object independently. Note that under this setting, for the three scenarios (i), (ii), and (iii), the comparisons between $rInt$ and $wInt$ are satisfied stochastically.

We present the results for a single value of $wInt = 4.3\ sec$ for write operations. For scenario (i) we use $rInt = 2.3\ sec$, for scenario (ii) we use $rInt = 4.3\ sec$, and for scenario (iii) we use $rInt = 6.3\ sec$. The results are given in Figure 4, setting a.

We observe that the results in this setting are similar, with the percentage of two-round reads is mainly affected by the number of faulty servers. In all cases the percentage of two-round reads is under 7.5%.

Setting b: Fixed interval simulations. In this setting the intervals between two read (or two write) operations are fixed at the beginning of the simulation. All readers use the same interval $rInt$, and the writer the interval $wInt$. This family of simulations represent conditions where operations can be frequent and bursty.

Figure 4, case b(i) illustrates the case of $rInt < wInt$, where $rInt = 2.3\ sec$. Here a read (write) operation is invoked by every reader (resp. writer) in the system every $rInt = 2.3\ sec$ (resp. $wInt = 4.3\ sec$). Because of

asynchrony not every read operation completes before the invocation of the write operation and thus we observe that only 4.5% of the reads perform two communication rounds.

Figure 4, case b(ii) illustrates the scenario where $rInt = wInt$. This is the most bursty scenario since all operations, read or write, are invoked at the same time, specifically the operations are invoked every $rInt = wInt = 4.3\ sec$. Although the conditions in this case are highly bursty (and unlikely to occur in practice), we observe that only about half of the read operations perform two communication rounds.

Figure 4, case b(iii) illustrates the scenario where $wInt < rInt$. In particular a read operation is invoked every $rInt = 6.3\ sec$ by each reader and a write operation every $wInt = 4.3\ sec$. Given the modeled channel latency and delays, notice that there is no concurrency between the read and write operations in this scenario. So all the servers reply to any read operation with the latest timestamp and thus no read operation needs to perform a second communication round.

Finally, note the common trend that increasing the number of readers and the number of faulty servers negatively impacts the performance of the algorithm in the scenarios (i) and (ii) for both case a and case b.

9 Conclusions and Future Work

In this paper we investigated the existence of semifast implementations of a read/write atomic register. It is shown in [3] that there are no fast SWMR implementations—where both readers and the writer perform one communication round—if there are $\frac{S}{t} - 2$ or more readers. Furthermore a question was posed whether there exist semifast implementations where reads or writes are fast.

The goal of this paper is to relax the bound on the readers in the system at the cost of allowing some reads to perform two communication rounds. We formalized the notion of semifast implementations and we presented an implementation that meets our goal and satisfies the required properties. For our implementation we show that for any write operation only one complete read operation (and maybe some read operations concurrent with that) needs to perform two communication rounds. We also showed that there is no semifast implementation if the number of different *virtual nodes* in the system is $\frac{S}{t} - 2$ or greater. Moreover we showed that there cannot exist semifast implementations for the MWMR model. Finally, we simulated our algorithm and presented the results that demonstrate that most read operations are fast in our simulated executions.

Our paper made progress in identifying the tradeoffs between the concurency in the system and the number of communication rounds required to implement atomic registers. The next step is to better understand the tradeoffs in the MWMR model. One direction is to consider hybrid semifast implementations where writers and readers perform a mixture of fast and semifast operations. Another direction is to consider dynamic settings such as [9] where nodes might join, leave, and arbitrarily fail. Lastly, more virulent adversarial behaviors, such as Byzantine failures, can be be studied and analyzed. Initial progress in this direction is reported in [6, 7], where certain quorums are

employed to trade operation latency and Byzantine-resilience in SWMR implementations. The broader question we intend to investigate is—given a particular distributed system model—how fast can a distributed atomic read be?

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. 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. *Distributed Computing*, 18(2):125–155, 2005.

[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*, pages 236–245, 2004.

[4] B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 454–463, 2000.

[5] C. Georgiou, N. Nicolaou, and A. Shvartsman. Fault-tolerant semifast implementations for atomic read/write registers. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–290, 2006.

[6] R. Guerraoui and M. Vukolić. How fast can a very robust read be? In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 248–257, New York, NY, USA, 2006. ACM.

[7] R. Guerraoui and M. Vukolić. Refined quorum systems. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 119–128, New York, NY, USA, 2007. ACM.

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

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

[10] 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*, pages 272–281, 1997.

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