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

Chryssis Georgiou†
University of Cyprus, Cyprus
chryssis@cs.ucy.ac.cy

Nicolas C. Nicolaou
University of Connecticut, USA
nicolas@engr.uconn.edu

Alexander A. Shvartsman
University of Connecticut and MIT, USA
aas@engr.uconn.edu

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 round-trip communication, and where reads complete in one to two communication rounds 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. 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-round 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.

*This work is supported in part by the NSF Grants 9988304, 0121277, and 0311368.

†The work of this author is supported in part by research funds at the University of Cyprus.

SPAA'06, July 30–August 2, 2006, Cambridge, Massachusetts, USA.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Algorithms, Reliability, Theory

Keywords

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

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 round-trip communication stage, while the read protocol involves two round-trip stages, where the second stage essentially performs the write of the value obtained in the first stage. Following this development, a folklore belief developed that in messaging-passing atomic memory implementations "atomic reads must write". However, recent work by Dutta *et al.* [3] established that if the number of readers is appropriately constrained with respect to the number of 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 larger 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.

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 pro-

cesses 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 responses 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 reads 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 [8, 4]. A fully dynamic atomic memory implementation using reconfigurable quorums is given in [7], 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 Geo-Quorums 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. 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.

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 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. More broadly, our contributions are as follows.

1. We define the notion of a *semifast* implementation which specifies which atomic reads are required to perform a second communication round. In particular, for each write operation, only *one complete* read operation is allowed to perform two communication rounds.
2. We provide a semifast implementation of an atomic read/write object that supports arbitrarily many readers. 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. 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.
4. We show that there does not exist semifast atomic implementations for multiple writers and multiple readers, even for $t = 1$.
5. We simulated our SWMR implementation and we present preliminary 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 10% of the read operations required a second round.

Paper Organization. In Section 2 we present our model and definitions. In Section 3 we describe our implementation and prove its correctness. In Section 5 we show the necessary properties that an implementation of an atomic register must possess in order to be semifast. In Section 6 we show that no semifast MWMR implementation is possible. Section 7 contains simulation results. Due to space limitations, some proofs are omitted, and they can be found in [5].

2. MODEL AND DEFINITIONS

We consider the single writer, multiple reader (SWMR) model, where a distinguished process w is the writer, the set of R readers are processes with unique ids from the set $\mathcal{R} = \{r_1, \ldots, r_R\}$, and where the object replicas are maintained by the set of S servers with unique ids from the set $\mathcal{S} = \{s_1, \ldots, s_S\}$ such that at most t servers can crash. 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].)

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 be-

tween 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*. We refer to the messages that intend to write a new value to the atomic register as WRITE messages, and we call the messages that request the value of the register as READ messages. The messages used to propagate information within the system are called INFORM messages.

An algorithm A is a collection of automata, where A_p is the automaton assigned to the process p with an initial state $Init$. 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 (with st_p denoting the state component of process p) and includes the set of messages $mset$ and the state of each process in 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. 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. In every step a process p acts as follows, where st' is the resulting state (the state components of all other processes are unchanged in st'): (1) it sets $st'.mset$ to $st.mset - mIn$, (2) inputs mIn, inv, and its current state st_p to A_p, which outputs a new state st'_p, the messages $mOut$ to be sent, and the response res to the last invoked operation, and (3) adopts the state st'_p as its new state, sets $st'.mset$ to $st.mset \cup mOut$, and responds with res to the application. A process p performs an *invocation step* if the invocation $inv \neq \perp$, a *response step* when $res \neq \perp$, and a *communication step* if $mOut \neq \emptyset$ and both $inv = \perp$ and $res = \perp$.

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 it begins with the step $\sigma_0 = \langle s0, *, *, *, *, *\rangle$ where $s0$ is the initial state of the system and $s0.mset = \emptyset$, and for each process p, $s0_p = Init$. Executions are denoted by the symbol ξ. A finite execution fragment φ is a finite prefix of some execution. We say that an execution fragment φ *extends* some finite execution fragment φ' if the first step in φ is $\sigma_f = \langle st', *, *, *, *, *\rangle$, the last step of φ' is $\sigma_\ell = \langle st, *, *, *, *, *\rangle$, such that st' is the state that immediately results from σ_ℓ.

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*.

Atomicity. Our goal is to implement a read/write atomic object in a message passing system by replicating the value of the object among the servers in the system. Each replica consists of a value v, initially $\perp$, and an associated timestamp ts, initially 0. A read or a write operation 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 a (read or write) operation 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. 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. Atomicity is defined as follows [6]: Consider the set Π of all complete operations in any well-formed execution. Then there exists an irreflexive partial ordering $\prec$ on operations in Π, satisfying the following: (1) For any operation $\pi \in \Pi$, there are finitely many operations π' such that $\pi' \prec \pi$. (2) If operation π_1 precedes the operation π_2 in Π, then it cannot be the case that $\pi_2 \prec \pi_1$. (3) If π is a write operation and π' is any operation in Π, then either $\pi \prec \pi'$ or $\pi' \prec \pi$. (4) The value returned by a read operation is the value written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write).

Semifast implementations. We say that a read or write operation π is *fast* if it completes in one communication round. Let denote by $inv(\pi)$ the invocation of operation π requested at process p and let $ret(\pi)$ denote the response of process p for operation π. We then define a communication round as follows:

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 an invocation step where $inv = inv(\pi)$ or a communication step during π, to a subset of processes,
(2) any process p' that receives $m \in mIn$ 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 messages $m' \in mIn$, it either performs a response step with $res = ret(\pi)$ or inserts a set of messages in $mOut$ and performs a communication step.

When process p decides to respond to the application within π in (3) above during the first communication round in π, then we say that operation π is *fast*. An implementation is *fast* if both reads and writes 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. Let w_k be the k^{th} $(k \geq 1)$ write operation by the sole writer and let val_k be the value written to the register. Let $\prec$ be the partial order defined on any (atomic) execution as given earlier. We use the reading-function $\Re(\rho)$ as defined in [9] to specify the (always unique) write operation that wrote the value returned by read ρ.

DEFINITION 2.2. *An SWMR implementation I is* semifast *if the following are satisfied:*
(1) In any execution ξ of I, every write *operation w_k, $k \geq 1$, 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 a two-round read operation ρ_1 returns the value $ret(\rho_1) = val_k$ and $\Re(\rho_1) = w_k$, then any read operation ρ_2, where $\rho_1 \prec \rho_2$ or $\rho_2 \prec \rho_1$, and $ret(\rho_2) = val_k$ and $\Re(\rho_2) = w_k$, must be fast.
(4) There exists an execution $\xi$of I containing a write operation w_k and a set of read operations $\mathcal{F}$ such that $\forall \rho \in \mathcal{F}, \Re(\rho) = w_k$ and ρ is fast.

[1]Notice that process p' replies to m 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 m.

We make the following observations having the above definition in mind. Given that any subset of the readers and the writer may fail, in order to guarantee termination, no operation can wait for replies from any reader or writer processes. 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 in the sequel 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 of a semifast implementation, 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. IMPLEMENTATION SF

We now present a semifast implementation, called SF, in which there is one writer and 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 5 that semifast implementations are possible iff $V < \frac{S}{t} - 2$). We now describe our implementation presented in pseudocode in Figure 1. Recall that each replica consists of a value and its timestamp. For simplicity we give the algorithm that returns only the timestamps; then we describe a straightforward modification that returns a value along with each timestamp.

Writer. During a write operation, the writer w sends a write message consisting of the current timestamp and the value to be written to all servers. Since t of the servers might be faulty, w waits for responses from only $S - t$ servers. Upon receipt of all the expected acknowledgments the writer increases its timestamp and completes the operation. The timestamps impose a natural order on the writes since there is only one writer.

Server. The servers maintain the replicas of the object. The state of a server includes the following: (1) ts the greatest timestamp received any server, (2) the set *seen* where the server records the *virtual ids* of the readers that read the latest timestamp of the server, (3) the *counter* array in order to distinguish new from old messages from each process (needed because of asynchrony), and (4) the variable *postit* used by readers to inform, if necessary, other readers 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. 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. Otherwise, if $ts' < ts$, s_i sets its *seen* set to be equal to $seen \cup \{\nu(p_j)\}$ declaring that p_j perceived 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 a reply to p_j acknowledging the transaction. If a READ, WRITE or INFORM message is received then the reply is a READACK, WRITEACK, or an INFORMACK, respectively. An INFORM message denotes that the process p_j wants to inform the rest of the reader processes about the timestamp it is about to return. Accordingly, before replying to an INFORM message, s_i updates its *postit* value. To ensure that the value enclosed in the INFORM message is not an already-returned timestamp, s_i compares the received timestamp with its *postit*. If the *postit* value is greater, it must be the case that another reader already returned a newer timestamp than the one in the message and so updating the *postit* with an older timestamp may violate atomicity; otherwise *postit* is updated. 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. When r_i invokes a read operation, it sends messages to all servers and waits for $S - t$ responses. Each of these responses is of the form (READACK, $ts', seen, rCounter, postit$). Upon collecting these messages, r_i checks $rCounter$ to distinguish new messages from the stale messages (due to asynchrony), and then records the maximum timestamp $maxTS = ts'$ and the maximum postit $maxPS = postit$ value contained among the received messages. Based on the received information, reader r_i computes the set of messages that contained the maximum timestamp ($maxTSmsg$). Then the following predicate is used to decide the return value: we check if there is a subset $MS \subseteq maxTSmsg$ such that its cardinality $|MS| \geq S - \alpha t$, for some $\alpha \in [1, V+1]$ and the cardinality of the intersection of the messages in MS is $|\cap_{m \in MS} m.seen| \geq \alpha$, then we return $maxTS$. The predicate can be interpreted as "enough processes have seen the $maxTS$ that we received".

In order to visualize the idea behind the predicate consider an finite execution fragment φ_1 where the writer w performs a complete write operation ω_1 which receives replies from $|S_{w(1)}| = S - t$ servers. We extend φ_1 by a complete read operation ρ_1 which misses t servers from those that responded to ω_1, such that $|MS_1| = |S_{w(1)} \cap S_1| = S - 2t$ where S_1 is the set of $S - t$ servers that responded to ρ_1 . According to atomicity, the read operation ρ_1 returns $TS_1 = maxTS$. Consider now another execution φ_2 where the write operation ω_1 is incomplete and receives replies from exactly $|S_{w(1)}| = S - 2t$ servers. We extend φ_2 with a read operation ρ_1 from r_i which receives replies from $|S_1| = S - t$ servers including the servers in $S_{w(1)}$. So $|MS_1| = |S_{w(1)} \cap S_1| = S - 2t$ and thus the read ρ_1 cannot distinguish execution φ_2 from φ_1. Hence, by atomicity, ρ_1 returns $TS_1 = maxTS$ in φ_2 as well. By extending φ_2 even further by a second read operation ρ_2 from r_j we might get into the situation where $|MS_2| = |S_{w(1)} \cap S_2| = S - 3t$, where $|S_2| = S - t$ the servers that responded to ρ_2. But in order to preserve atomicity the reader r_j must also return $TS_2 = maxTS$. This scenario can be easily generalized for more than two read operations and so the predicate in line 23 of the algorithm in Figure 1 arise to preserve atomicity between the different read operations.

Note here that the above result is true if the unique ids of the readers are recorded in the *seen* set. If we record the virtual ids of r_i and r_j (as it is done in our implementation) we only get the same result if the two readers are not siblings. In different case, namely where $\nu(r_i) = \nu(r_j) = \nu_k$, the *seen* set witnessed by both ρ_1 and ρ_2 in ξ_2 could be $|\cap_{m \in MS_1} m.seen| = |\cap_{m \in MS_2} m.seen| = |\{w, \nu_k\}| = 2$. If so the predicate would not hold for ρ_2, returning $maxTS - 1$ and violating atomicity.

at the writer w
procedure initialization:
$ts \leftarrow 1,\ rCounter \leftarrow 0$
procedure write(v)
$rCounter \leftarrow rCounter + 1$
send($WRITE,\ ts,\ rCounter, 0$) to all servers
wait until receive($WRITEACK,\ ts,\ *,\ rCounter,\ *$) from $S - t$ servers
$ts \leftarrow ts + 1$
return(OK)

at each reader r_i
procedure initialization:
$vid(r_i) \leftarrow (i \bmod (\frac{S}{t} - 2) + 1),\ ts \leftarrow 0,\ rCounter \leftarrow 0,\ maxTS \leftarrow 0,\ maxPS \leftarrow 0$
procedure read()
$rCounter \leftarrow rCounter + 1$
$ts \leftarrow maxTS$
send($READ,\ ts,\ rCounter,\ vid(r_i)$) to all servers
wait until receive($READACK,\ *,\ *,\ rCounter,\ *$) from $S - t$ servers
$rcvMsg \leftarrow \{m | r_i$ received $m = (READACK,\ *,\ *,\ rCounter,\ *)\}$
$maxTS \leftarrow$ **Maximum**$\{ts' | (READACK,\ ts',\ *,\ rCounter,\ *) \in rcvMsg\}$
$maxTSmsg \leftarrow \{m | m.ts = maxTS$ and $m \in rcvMsg\}$
$maxPS \leftarrow$ **Maximum**$\{postit | (READACK,\ *,\ *,\ rCounter, postit) \in rcvMsg\}$
$maxPSmsg \leftarrow \{m | m.postit = maxPS$ and $m \in rcvMsg\}$
if there is $\alpha \in [1, V + 1]$ and there is $MS \subseteq maxTSmsg$ s.t. $(|MS| \geq S - \alpha t)$ **and** $(|\cap_{m \in MS}\ m.seen| \geq \alpha)$ **then**
if $|\cap_{m \in MS}\ m.seen| = \alpha$ **and** $(maxPS < maxTS$ **or** $|maxPSmsg| < t + 1)$ **then**
send(*INFORM*, $maxTS,\ rCounter,\ vid(r_i)$) to $3t + 1$ servers
wait until receive(*INFORMACK*, $*,\ *,\ rCounter,\ *$) from $2t + 1$ servers
end if
return($maxTS$)
elseif $maxPS = maxTS$ **then**
if $|maxPSmsg| < t + 1$ **then**
send(*INFORM*, $maxTS,\ rCounter,\ vid(r_i)$) to $3t + 1$ servers
wait until receive(*INFORMACK*, $*,\ *,\ rCounter,\ *$) from $2t + 1$ servers
end if
return($maxTS$)
else
retutn($maxTS - 1$)
end if

at each server s_i
procedure initialization:
$ts \leftarrow 0,\ seen \leftarrow \emptyset,\ counter[0...R] \leftarrow 0,\ postit \leftarrow 0$
procedure serve()
upon receive($msgType,\ ts',\ rCounter',\ vid$) from $q \in \{w, r_1, \ldots, r_R\}$ **and** $rCounter' \geq counter[pid(q)]$ **do**
if $ts' > ts$ **then**
$ts \leftarrow ts'; seen \leftarrow \{vid\};$
else
$seen \leftarrow seen \cup \{vid\}$
end if
$counter[pid(q)] \leftarrow rCounter'$ /* $pid(q)$ returns 0 if $q = w$ and i if $q = r_i$ * /
if $msgType =$*READ*
send(*READACK*, $ts,\ seen,\ rCounter',\ postit$) to q
else if $msgType =$*WRITE*
send(*WRITEACK*, $ts,\ seen,\ rCounter',\ postit$) to q
else if $msgType =$*INFORM*
if $postit < ts'$ **then**
$postit \leftarrow ts'$
end if
send(*INFORMACK*, $*,\ *,\ rCounter',\ postit$) to q
end if

Figure 1: Implementation SF

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.

Remark: By the above implementation, if all readers form one virtual node ($V=1$), then a read operation ρ will return $maxTS$ only when it receives at least $S-2t$ replies which contain $maxTS$. But this implies that the write operation which wrote $maxTS$ must be either completed or requires at most t more replies to complete. Consequently in order to achieve efficiency, it is important to study the division of the reader processes among the virtual nodes. This is left as an open question.

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 associated values and if it decides (as before) 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 (that of the preceding write).

We now give the correctness of algorithm SF.

THEOREM 3.1. *Algorithm SF implements a semifast atomic SWMR read/write register.*

PROOF. (Sketch.) The proof is done in two parts. We first show that SF implements an atomic read/write register in the SWMR model by showing that in any execution the atomicity properties are not violated (see Section 4). Then we show that SF is a semifast implementation by showing that the requirements of Definition 2.2 are met. □

4. CORRECTNESS OF SF

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: (1) If a read operation returns, it returns a non-negative integer, (2) if a read ρ is complete and succeeds some write(k), then ρ returns ℓ such that $\ell \geq k$, (3) if a read ρ returns $k (k \geq 1)$, then write(k) either precedes ρ or is concurrent with ρ, (4) 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.

Before proceeding to the proof we first introduce some notation we use throughout this section. Each read operation is denoted by ρ_i. For each read operation ρ_i, let S_i denote the set of servers that received messages from ρ_i and replied to those messages. For the writer we denote the set of servers that received messages from the k^{th} write operation as $S_{w(k)}$. Furthermore let $MaxS_i$ be the set of servers that replied with the maximum timestamp to ρ_i, and therefore $MaxS_i \subseteq S_i$. The set of messages received from ρ_i containing the maximum timestamp and sent by the servers in $MaxS_i$, is represented by MS_i. The maximum timestamp received by the read ρ_i is represented as TS_i. If a read operation ρ_i performs a second communication round, then we denote as NS_i to be the set of servers that received the messages from the second communication round of ρ_i and replied to those messages. 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. 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.

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 $\langle r_i, \nu_* \rangle$ and $\langle r_j, \nu_* \rangle$ perform subsequent reads ρ_1 and ρ_2, respectively. Then, for any execution ξ of SF, $| |MaxS_1| - |MaxS_2| | \leq t$.*

The proofs of the first and third atomicity conditions (as given above) are omitted because of their triviality.

LEMMA 4.2. *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 ξsuch that $\sigma < \sigma'$ and $ts_{s_i} = y$, we have that $y > x$.*

PROOF. This can be ensured by line 44 of Figure 1. □

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

LEMMA 4.3. *In any execution $\xi$of SF, if a server s_i sets its $postit_{s_i}$ to x at a step σ, then, given any step σ' of ξsuch that $\sigma < \sigma'$ and $postit_{s_i} = y$, we have that $y > x$.*

PROOF. This can be ensured by line 55 of Figure 1. □

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

PROOF. Consider an execution $\xi$of SF where the read operation ρ_1 introduced a postit equal to y to the system. It follows that ρ_1 observed as the maximum timestamp in the system $TS_1 = x$. As $|MS_1| \geq S - \alpha t$ and $|\cap_{m \in MS_1} m.seen| = \alpha$, ρ_1 performs an informative operation. Since $\alpha \in [1, V+1]$ and $S > (V+2)t$, we get that $|MS_1| > t$. So, if we denote by S_2 the set of servers that replied to a subsequent read ρ_2 ($|S_2| = S - t$), then per Lemma 4.2 there is a server, $s_i \in MaxS_1 \cap S2$ that replies to ρ_2 with a timestamp $ts' \geq x$. Therefore, ρ_2 will detect a maximum timestamp $TS_2 \geq ts'$, and hence $TS_2 \geq x$. □

LEMMA 4.5. *For any execution $\xi$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 $\xi$of SFwhich contains a read operation ρ_1 by a reader $\langle r_i, \nu_i \rangle$. It follows from Lemma 4.4 that if read ρ_1 receives a $postit = x$, then it will detect a maximum timestamp $TS_1 \geq x$. Let $TS_1 = x$ and so either the predicate will hold and then ρ_1 will return $y = TS_1$, or the condition whether $postit_{r_i} = TS_1$ will be true and so ρ_1 will in this case return

$y = TS_1$ as well. Thus ρ_1 will return $y = x$. If now $TS_1 > x$ then ρ_1 will return $y = TS_1$ if the predicate holds or $y = TS_1 - 1$ otherwise. Note that since $postit = x$, it is less than TS_1 and so the postit condition does not hold. Either case ρ_1 will return a value $y \geq x$. □

The following lemma ensures the second atomicity property.

LEMMA 4.6. *For any execution ξ of SF, if a read ρ_1 is complete and succeeds some write(k), then ρ_1 returns ℓ such that $\ell \geq k$.*

PROOF. Suppose that the writer w performs a $write(k)$ operation and precedes the read ρ_1 operation by reader r_i with virtual id ν_i during an execution $\xi$of SF. Let S_w be the $S - t$ servers that replied to w in the same execution. The intersection between S_w and S_1, $MaxS_1 = S_w \cap S_1$, is obviously $|MaxS_1| \geq S - 2t$. Since wr preceded ρ_1 the timestamp ts for each server in $MaxS_1$, per Lemma 4.2 it is greater or equal to k. So ρ_1 received a maximum timestamp TS_1 such that $TS_1 \geq k$. From the implementation we know that the reader returns either TS_1 or $TS_1 - 1$. We consider two cases:
Case 1: $TS_1 > k$. Since ρ_1 returns either TS_1 or $TS_1 - 1$, it follows that either case it returns a timestamp greater or equal to k.
Case 2: $TS_1 = k$. As we mentioned above each server in $MaxS_1$ replies with a $ts \geq k$. Since $TS_1 = k$ every server $s_i \in MaxS_1$ replies with a timestamp $ts = k$ to ρ_1. So the set MS_1, which contains the messages received by ρ_1 with the highest timestamp, will include the messages sent by all the servers in $MaxS_1$. So $|MS| \geq S - 2t$. 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 $MaxS_1$. Before the servers in $MaxS_1$ 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 $TS_1 = k$. Observe that no reader will return TS_1 because of a postit in the system because the predicate will hold for every process in the system for $\alpha = 2$, since the writer w has no sibling processes. □

In order to prove the forth atomicity property, we first need to show that readers who belong to the same virtual node (siblings) satisfy that property. Then we show that the property is also true for any two non-sibling readers in the system.

LEMMA 4.7. *Let the readers $\langle r_j, \nu_k \rangle$ and $\langle r_i, \nu_k \rangle$ be siblings and perform the read operations ρ_1 and ρ_2 respectively. For any execution $\xi$of SF that contains ρ_1 and ρ_2, if ρ_1 precedes ρ_2, and ρ_1 returns x then ρ_2 returns y, such that $y \geq x$.*

PROOF. Consider an execution $\xi$of SF. Let first investigate the case where $r_j = r_i$. In this case ρ_1 denotes the first read operation of r_j 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_j sends a READ message with $ts_{r_j} = TS_1 \geq x$. This message will be received by all servers in S_2 which according to Lemma 4.2 will reply with a timestamp $ts' \geq TS_1 \geq x$. So $TS_2 \geq x$. If $TS_2 = x$ then $|MS_2| = S - t$ and the predicate holds for $\alpha = 1$. Thus $y = TS_2 = x$. Otherwise, if $TS_2 > x$, the return value y will be equal to TS_2 or $TS_2 - 1$ 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 $r_j \neq r_i$. We investigate the following two possible cases: (1) ρ_1 returns $x = TS_1 - 1$ and (2) ρ_1 returns $x = TS_1$. In all of the cases we show that $x \leq y$ or that the case is impossible.

Case 1: In this case $x = TS_1 - 1$. Therefore, some servers replied to ρ_1 with $TS_1 = x + 1$, and hence a write($x + 1$) operation had started before ρ_1 is completed. So write(x) completed before ρ_1 has completed and moreover before ρ_2 is executed since ρ_1 precedes ρ_2. Thus by Lemma 4.6 ρ_2 returns a value $y \geq x$.

Case 2: In this case $x = TS_1$. Hence either there is some $\alpha \in [1, V + 1]$ such that $|MS_1| \geq S - \alpha t$ and $| \cap_{m \in MS_1} m.seen| \geq \alpha$ or ρ_1 received a $postit$ equal to TS_1 from some server. We examine those two possibilities separately.

Case 2(a): It follows that $x = TS_1$, and there is some $\alpha \in [1, V + 1]$ such that MS_1 consist at least $S - \alpha t$ messages received by ρ_1 with $ts = x$ and $| \cap_{m \in MS_1} m.seen| \geq \alpha$. Since $V < \frac{S}{t} - 2$ and $a \in [1, V + 1]$, then $|MS_1| = S - at > t$. We have two cases to consider for ρ_1: (1) First let examine the case where ρ_1 returns $x = TS_1$ because $| \cap_{m \in MS_1} m.seen| = \alpha$. According to the implementation, ρ_1 has to inform $|NS_1| \geq 2t+1$ servers about its return value, x. Since ρ_1 precedes ρ_2, at least $|NS_1 \cap S_2| \geq 2t + 1$ servers, that informed by ρ_1, will reply to ρ_2. Any server $s_i \in NS_1 \cap S_2$, by Lemma 4.4 will reply with a $postit \geq x$ to ρ_2 and with a timestamp $ts \geq x$. So ρ_2 will observe a maximum timestamp $TS_2 \geq x$. According now to Lemma 4.5 ρ_2 will return a value $y \geq x$. (2) The second case arise when ρ_1 returns $x = TS_1$ because $| \cap_{m \in MS_1} m.seen| > \alpha$. 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 $y = TS_2$ or $y = TS_2 - 1$:
(i) Let first consider the case where $y = TS_2$. Since ρ_1 returned $x = TS_1$, as we mentioned in (1) , there is a write(x) operation that preceded or was concurrent with ρ_1. As stated above $|MS_1| > t$ and hence there is a server s_i such that $s_i \in MaxS_1 \cap S_2$. By Lemma 4.2, s_i will send a timestamp $ts \geq x$ to ρ_2, and hence $TS_2 \geq ts$. So $y \geq x$.
(ii)We now get down to the case where ρ_2 returns $y = TS_2 - 1$. Since $|MaxS_1| > t$, there must be a server $s_i \in MaxS_1 \cap S_2$ and s_i replies with a timestamp $ts \geq x$ to ρ_2. So the highest timestamp in S_2(i.e. $TS_2 = y + 1$) 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, $TS_2 = y+1 = x$. Hence all the servers in $MaxS_1 \cap S_2$ replied with a timestamp $ts = x = y + 1$ to ρ_2. Recall that this case arise only when $| \cap_{m \in MS_1} m.seen| > \alpha$. Also according to Lemma 4.1, $||MS_2| - |MS_1|| \leq t$ and hence $|MS_2| \geq S - (\alpha + 1)t$. For any $s_i \in MaxS_1 \cap S_2$, we denote as m_1 the message sent by 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 m_1 sent before m_2 then $m_1.seen \subseteq m_2.seen$. As a result $| \cap_{m \in MS_1} m.seen| \leq | \cap_{m \in MS_2} m.seen|$. 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 $| \cap_{m \in MS_1} m.seen| = | \cap_{m \in MS_2} m.seen|$. Either case, $| \cap_{m \in MS_2} m.seen| > \alpha$ and hence $| \cap_{m \in MS_2} m.seen| \geq \alpha + 1$. Observe that the predicate now is true for $\alpha + 1$ since $|MS_2| \geq S-(\alpha+1)t$, and thus ρ_2 must return $TS_2 = x = y+1$, contradicting the initial assumption that $y = x + 1$. 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 $x = TS_1$ because there was not $\alpha \in [1, \ldots, V + 1]$, such that $| \cap_{m \in MS_1} m.seen| \geq \alpha$, but some postits equal to TS_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_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 $postit = TS_1 = x$, then the writer w initiated a write(x) operation during or before ρ_1 completed. It follows that $NS_{1a} \cap S_1$ denote the set of servers that replied to ρ_1 and contained the $postit = TS_1$. The reader ρ_2 receives replies from $|S_2| = S - t$ servers. Since $|NS_{1a} \cap S_1| \geq t+1$, then $|S2 \cap (NS_{1a} \cap S_1)| \geq 1$. So the read operation ρ_2 will receive a reply from at least one server $s_i \in NS_{1a} \cap S_1$. Hence, from Lemma 4.3, ρ_2 receives a $postit \geq x$ from s_i and according to Lemma 4.5 will return a value $y \geq postit$ and thus $y \geq x$.
(2) Let now examine if ρ_1 received less than $t+1$ messages containing postits with value equal to TS_1. Let assume again that $|NS_{1a} \cap S_1| < t+1$ is the set of servers that replied with $postit = TS_1$ to ρ_1. However, in contrary to the previous case, the situation where $|(NS_{1a} \cap S_1) \cap S2| = 0$ might arise. So r_i gets into the information process in order to inform sufficient servers about its potential return timestamp. So at the time where ρ_1 is completed, $|NS_1| \geq 2t+1$ servers contain a $postit \geq TS_1$. When ρ_2 is performed, TS_2 is greater than or equal to x, since there is a server $s_i \in MS_1 \cap S_2$ and, according to Lemma 4.4, s_i returns a timestamp $ts \geq x$. Furthermore there is a server $s_j \in NS_1 \cap S_2$, and so according to Lemma 4.3, s_j replies with a $postit \geq x$. So, by Lemma 4.5, ρ_2 returns a value $y \geq x$. □

Similarly we proof that the fourth atomicity properties is also satisfied for any two non-sibling reader processes in the system.

LEMMA 4.8. *Let the readers $\langle r_j, \nu_j \rangle$ and $\langle r_i, \nu_i \rangle$ be non-siblings and perform the read operations ρ_1 and ρ_2 respectively. For any execution $\xi$of SF that contains ρ_1 and ρ_2, if ρ_1 precedes ρ_2, and ρ_1 returns x then ρ_2 returns y, such that $y \geq x$.*

THEOREM 4.9. *Algorithm SF implements an atomic read/write register in the SWMR model.*

PROOF. It follows from the fact that every process guarantees termination by waiting for only $S-t$ replies and the lemmas proved above. □

5. IMPOSSIBILITY

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) do not use any grouping assumptions and thus consider each node acting individually in the system, and (2) consider grouping mechanisms such as in algorithm SF. In our context by "grouping mechanism" we only mean the grouping of the reader processes in any arbitrary fashion. In other words we omit the grouping techniques that involve grouping of non-reader processes in the system. Additionally, Lemma 5.2, shows that informing at least $3t+1$ servers during a second communication round is a tight bound for any semifast implementation.

In algorithms where there is no grouping mechanisms assumed we can consider each reader to form an individual group. So the number of virtual nodes V is equal to the number of readers R. As showed in [3] in such systems there is no fast implementation of the read/write register if $R \geq \frac{S}{t} - 2$. However this violates the fourth property of the semifast definition and thus no such systems can be semifast. Hence our bound applies in these kind of systems. We now show the following considering algorithms using a grouping mechanism similar to SF:

LEMMA 5.1. *No semifast implementation exists if the number of node groups V in the system is $\geq \frac{S}{t} - 2$.*

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

We now state the main result of this section.

THEOREM 5.3. *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 5.1 and 5.2. □

6. MWMR MODEL

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

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 2.2). First we extract several immediate properties from the definition of atomicity presented in Section 2. If for given operations π_1 and π_2 in an execution, the response step of π_1 precedes the invocation step of π_2, we denote this by $\pi_1 \rightarrow \pi_2$. 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 wr that writes value v and a read operation ρ_i such that $wr \rightarrow \rho_i$, and all other writes precede wr then ρ_i returns v. 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, 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 from the system at that round. We say that an operation performs a *write phase* during a communication round if it propagates information to other participants 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 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 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 these 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 $r_i(j)$ the j^{th} communication round of a read operation from reader r_i. An arbitrary delay may occur between two communication rounds $r_i(j)$ and $r_i(j+1)$ 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 $\pi \in sr_i(j-1)$, $\pi \rightarrow r_i(j)$. A set $sr_i(j-1)$ might be equal to the empty set containing no operations.

CLAIM 6.1. *A read operation ρ that succeeds any write operation ω or write phase ωp of an operation $\pi \neq \rho$, returns the value decided by the read phase preceding its last write phase.*

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. 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 $read_1()$ by r_1. These operations are not concurrent and they are executed in the order $write(2) \rightarrow write(1) \rightarrow read_1()$. By property P2, operation $read_1()$ 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 $read_1()$ by r_1. As before, these operations are not concurrent. So in this case, by property P2, operation $read_1()$ 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 $r_1(i)$ of the $read_1()$ 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 the n^{th} communication round, by property P2, the operation $read_1()$ 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 $r_1(i)$ of the $read_1()$ operation from reader r_1. In this case by the end of the n^{th} communication round of r_1, and by property P2, the $read_1()$ operation returns 2.

If we assume now, without loss of generality, that the last communication round $r_1(n)$ of r_1 in φ_{1g} is a write phase, 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 $r_1(i)$ of the $read_1()$ 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 $r_1(n)$. 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 6.1 and so by the same claim the decision for the return value must be made in $r_1(n-1)$. Notice that the decision of r_1 taken in $r_1(n-1)$ is not affected from the operations in $sr(n-1)$. 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 return 1 by the end of $r_1(n-1)$ in φ_{1g} and 2 by the end of $r_1(n-1)$ 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 6.2. *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.*

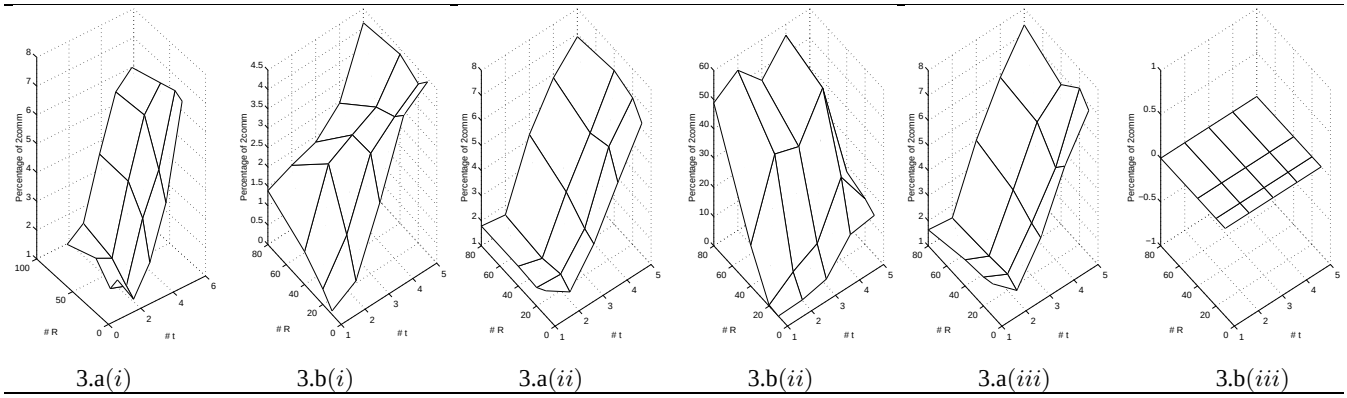
PROOF. The proof follows by reasoning on the construction presented above. See [5] for full details. □

7. SIMULATION RESULTS

To evaluate the effectiveness of our implementation, we simulated algortihm SF using the NS2 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. The testbed of our simulations included 20 servers out of which 5 may fail at arbitrary times. Since we require that $V < \frac{S}{t} - 2$, in order to maintain at least one group we can tolerate up to $\frac{S}{4}$ faulty servers. The number of reader processes varies between 10 and 80. We use $rInt$ and $wInt$ to stand for the time intervals between each read and write operations respectively. Several scenarios were tested: (i) frequent reads and infrequent writes, where $rInt < wInt$, (ii) concurrent reads and writes, such that $rInt = wInt$, and (iii) infrequent reads and frequent writes, such that $rInt > wInt$. The processes send their messages after a random delay to model asynchrony. 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).

Stochastic simulations. This is the 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 invocations renders a more realistic scenario where processes are interacting with the atomic object independently. Under this setting, for the three scenarios (i), (ii), and (iii), the comparisons between $rInt$ and $wInt$ may be satisfied only stochastically. A single value of $wInt = 4.3$ sec was chosen for the upper limit of any write operation. For the read operations the values of $rInt = 2.3$ sec, $rInt = 4.3$ sec, and $rInt = 6.3$ sec were chosen, with the results presented in Figure 2, set a. The results for this family of executions are similar where 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%.

Fixed interval simulations. Here the intervals for each read (or write) operation 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. The intervals $rInt$ and $wInt$ when $rInt \neq wInt$ are chosen to avoid having read operations invoked at the same time with write operations. In Figure 2, b(i) illustrates the case of $rInt < wInt$. A read (write) operation is invoked by every reader (resp. writer) in the system every $rInt = 2.3$ sec



3.a(i) 3.b(i) 3.a(ii) 3.b(ii) 3.a(iii) 3.b(iii)

Figure 2: Stochastic simulation 3.a; Fixed interval simulation 3.b. The vertical axes show the percentage of two-round reads as a function of the number of readers and the number of faulty servers.

(resp. $wInt = 4.3\ sec$). Because of asynchrony not every read operation completes before the invocation of the write operation and thus we observe a small percentage of reads that perform two communication rounds. In b(ii) the condition where $rInt = wInt$ is illustrated. This is the worst case scenario since all operations, read or write, are invoked at the same time, that is they are invoked every $rInt = wInt = 4.3\ sec$. Although the conditions in this case are highly adversarial, we observe that only about half of the read operations perform two communication rounds. Lastly, in b(iii) we study the case where $wInt < rInt$. In particular a read operation is invoked every $rInt = 6.3\ sec$ by each reader and a write operation every $wInt = 4.3\ sec$. In this case all write operations complete before any invocation step of a read operation. 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 first two scenarios.

8. 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 between two write operations only one complete read operation 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 [7] where nodes might join, leave and arbitrarily fail. The broader question we intend to investigate is—given a particular distributed system model—how fast can a distributed atomic read be?

Acknowledgment: We thank Rachid Guerraoui for his helpful comments.

9. 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, 2003.

[3] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 236–245. ACM Press, 2004.

[4] B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *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, 2005. http://www.cse.uconn.edu/ ncn03001/pubs/TRs/GNS06.pdf.

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

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

[8] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

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