How Fast can a Distributed Atomic Read be?

Partha Dutta Rachid Guerraoui Ron R. Levy Arindam Chakraborty

Distributed Programming Laboratory, EPFL
CH-1015, Lausanne, Switzerland

Abstract

This paper addresses the problem of designing an efficient implementation of a basic atomic read-write data structure over an asynchronous message-passing system. In particular, we consider time-efficient implementations of this abstraction in the case of a single writer, multiple readers (also called a SWMR atomic register) and S servers: the writer, the readers, and t out of the S servers may fail by crashing. Previous implementations tolerate the failure of any minority of servers (i.e., $t < S/2$) and require one communication round-trip for every write, and two round-trips for every read.

We investigate the possibility of *fast* implementations, namely, implementations that complete both reads and writes in *one* round-trip. We show that, interestingly, the existence of a fast implementation depends on the maximum number of readers considered. More precisely, we show that a fast implementation is possible *if and only if* the number of readers is less that $\frac{S}{t} - 2$. We also show that a fast implementation is *impossible* in a multiple writers setting when $t \geq 1$.

Our results draw sharp lines between the time-complexity of regular and atomic register implementations, as well as between single-writer and multi-writer implementations. The results lead also to revisit, in a message-passing context, the folklore theorem that "atomic reads must write".

1 Introduction

The atomic read-write data structure is one of the most fundamental abstractions in distributed computing. In short, this abstraction allows concurrent and crash-prone processes to share information through a common variable, as if they were accessing it in a sequential and failure-free manner. We consider the implementation of this abstraction in the case of a single-writer multi-reader, also called a SWMR *atomic register* [6], in an asynchronous message-passing system where processes may fail by crashing. Fault-tolerance is achieved by replicating the register over S servers. We seek implementations that tolerate the crash of the writer, of any subset of the readers and of any t out of S server processes.

1.1 Motivation

Informally, atomicity requires that, even though each read or write operation may take an arbitrary period of time to complete, they appear to be instantaneous at some point in time, during their respective period of execution [6]. This requires ordering operations in a way that they respect their temporal order as well as the sequential specification of a read-write data structure, namely, a read should return the last value written. The original message-passing implementation of [1] maintains the required order among operations by associating timestamps with every written value. To write some value v, the writer increments its local timestamp, and sends v with the new timestamp ts to all processes. (In [1], readers and servers are the same set, the writer is one of the servers, and a minority of processes may crash.[1]) Every process, on receiving such a message, stores v and ts and then sends an acknowledgment (an ack) to the writer. On receiving acks from a majority, the writer terminates the write. In a read operation, the reader first gathers value and timestamp pairs from a majority of processes, and selects the value v with the largest timestamp ts. Then the reader sends v and ts to all processes, and returns v on receiving acks from a majority of processes.

Roughly speaking, the ordering is ensured by associating the value (that is written or read) to the largest timestamp in the system, and then storing it at a majority of processes. Observe that, since we are in a single-writer setting, and since only the writer introduces new timestamps in the system, the writer always knows the latest timestamp. Thus, on invoking a write operation, the writer just needs to increment its own timestamp to get a timestamp that is higher than any existing timestamp in the system. On the other hand, a priori, a reader does not know the latest timestamp in the system, and hence, needs to spend one communication round-trip to know the latest value, and then another round-trip to propagate the value to a majority of processes.

The second round-trip is "required" in the above algorithm because the latest value learned in the first round-trip might be present at only a minority of processes. In a sense, every read includes, in its second communication round-trip, a "write phase", with the input parameter being the value selected in the first round-trip. This observation is captured by the folklore theorem that "atomic reads must write", which is actually borrowed from similar results in the shared-memory model [2, 6]. In particular, a theorem from [2] states that, to simulate a multi-reader atomic register from single-reader atomic registers, at least one of the readers must write into some single-reader register. Along the same lines, when implementing atomic registers over weaker *regular* ones [6], a process that reads a value v also needs to write it, in order to make sure that no other process will subsequently read an *older* value v': with a regular register, even if a value v' is written before a value v, v might be read before v', which is impossible with an atomic register. Recently, [3] has shown that, in a message-passing system, every atomic read must modify the state of at least t servers. However, in such a system, any message received by a server can potentially modify the server's state. Hence, a read can modify at least $S - t > t$ servers (assuming a majority of correct servers) in one round-trip. Thus [3] does not answer the question whether the

[1]The implementation also considers link failures, which can be masked by a retransmission module that does not increase time complexity in the absence of link failures.

second round-trip is necessary. In fact, processes (servers) are smarter than basic (regular or single-reader) registers and might intuitively do more. Hence the motivation for this work: to determine how fast an atomic read can be in a message-passing system.

In fact, we can reduce the time-complexity of the reads in [1] by using a simple decentralization combined with a *max-min* technique. First, the reader sends messages to all servers. Every server, on receiving such a message, broadcasts its timestamp to all servers. On receiving timestamps from a majority of servers, every server selects the *maximum* timestamp, adopts the timestamp and its associated value, and sends the pair to the reader. On receiving such messages from a majority of servers, the reader returns the value with the *minimum* timestamp.[2] But can we do better? Is there a *fast* implementation where none of the operations (read or write) requires more than one communication round-trip? This would clearly be optimal in terms of time-complexity. For example, consider the case of a fast implementation with two readers[3] and $t < S/2$. To illustrate this case, suppose the writer writes v with timestamp 7, and the write message is received only by one server s. (The write is incomplete.) The first reader gets information from a majority of servers that includes s. The read must return v because the reader does not know whether the write of v is complete or not, and this reader has to return the value of the last preceding write. Now suppose the second reader invokes a read, queries a majority of servers, and misses s. Clearly, the second read returns a value with a timestamp lower than 7, violating atomicity: the second read returns an older value than the preceding read. At first glance, it seems impossible to have a fast implementation with two readers. But what if we tolerate fewer faulty servers?

1.2 Contributions

We show that, interestingly, the existence of a fast SWMR implementation depends on the maximum number R of readers. More precisely, the primary contribution of this paper is to show that there is a fast implementation of a SWMR register if and only if $R < \frac{S}{t} - 2$.

1. Our fast implementation is based on the following observation: if a reader sees the latest timestamp ts at x servers, then any subsequent reader sees ts or a higher timestamp at $x - t$ servers; this is because, in a fast implementation, the first reader does not propagate ts, and the second reader might miss t servers seen by the first reader. A generalization of this observation helps determine when some reader can safely return the value associated with the latest timestamp. This is not entirely trivial because the

[2]To see why this ensures atomicity, observe that, when a write completes, its timestamp, say ts, is stored at a majority of servers. In any subsequent read, every server sees a timestamp that is higher than ts, before the server sends the message to the reader. Hence, the read returns a value that is not older than the written value. On the other hand, if a read returns a value with timestamp ts, then a majority of servers have a timestamp not lower than ts, and no subsequent read returns an older value.

[3]In the case of a single-reader and where a minority of servers may crash ($t < S/2$), it is trivial to modify the algorithm of [1] such that the read takes only one round-trip, i.e., does not “write”: the read can return the latest value learned from the servers in the first round trip, provided it is not older than the value returned in the previous read. Otherwise, the reader returns the same value as in the previous read. Since there is only one reader, this clearly orders the reads in the desired fashion and ensures atomicity.

safety of a value can not be simply deduced from the number of servers that has seen the value. To determine whether a value is safe, we have every server maintain, besides the latest value, the set of readers to which the server has sent that value.

2. Given S and t, we prove by contradiction that there is no fast implementation with $R \geq \frac{S}{t} - 2$. Given a fast implementation with $R \geq \frac{S}{t} - 2$, we consider a partial run which contains a write(1) that misses t servers, and we append it with a read that misses t other servers. Then we delete all the steps in the partial run that are not "visible" to the reader (basically, the steps of the t servers that the read missed). Still, the read returns 1 in the resulting partial run. Now we iteratively append reads by distinct readers, and delete the steps in the partial run that are not visible to the last reader, until we exhaust all the readers. To ensure atomicity, the last read of each partial run returns 1. In the final partial run (obtained after exhausting all readers) the steps of write(1) are almost deleted. We modify this partial run to construct several additional partial runs, one of which violates atomicity.

Our result determines the exact conditions under which "atomic reads must write" in a message-passing system, and draws a line between the time-complexity of regular and atomic register implementations. Indeed, whereas there is a fast implementation of an SWMR regular register if and only if $t < S/2$, irrespective of the number of readers (as long as this number is finite), our result states that a fast implementation of a SWMR atomic register exists if and only if $t < S/(R+2)$. Furthermore, we ask whether it is possible to have a fast implementation of a multi-writer multi-reader (MWMR) atomic register. We show that the answer is "no" if $t \geq 1$, and hence, we draw a line between the time-complexity of SWMR and MWMR atomic register implementations as well.

1.3 Roadmap

The rest of the paper only considers atomic registers and is organized as follows. Section 2 sketches the system model and the necessary definitions. We present a fast implementation of SWMR register assuming $R < \frac{S}{t} - 2$ in Section 3. We prove $\frac{S}{t} - 2$ to be a tight bound for R in Section 4. Section 5 considers the multi-writer case. We conclude the paper by discussing some open problems in Section 6.

2 Model and Definitions

2.1 Basics

The distributed system we consider consists of three *disjoint* sets of processes: a set *servers* of size S containing processes $\{s_1, ..., s_S\}$, a set *writer* containing a single process $\{w\}$,[4] and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. Associated with each process p is

[4]We discuss the multi-writer case in Section 5.

an application which asks p to invoke an operation, and p in turn gives the response of an operation back to the application. Every pair of processes communicates by message-passing using a bi-directional reliable communication channel. The state of communication channels is viewed as a set *mset* of messages, containing messages that are sent but not yet received. We assume that every message has two tags which identify the sender and the receiver of the message.

A distributed algorithm A is a collection of automata, where A_p is the automaton assigned to process p. Computation proceeds in *steps* of A which is denoted by an ordered tuple $< p, M, inv >$ consisting of a process id p, a set M of messages received in that step, and any invocation *inv* requested by the application in that step. (M might be $\emptyset$, and if there are no invocation request by the application in that step, then *inv* is $\perp$.) In step $< p, M, inv >$, process p atomically does the following: (1) remove the messages in M from *mset*, (2) apply M, *inv*, and its current state st_p to A_p, which outputs a new state st'_p, a set of messages to be sent, and (possibly) a response for the last invoked operation, and then (3) adopt st'_p as its new state, put the output messages in *mset*, and (possibly) respond back to the application. A step in which $inv \neq \perp$ is called an *invocation step*, and a step in which a process responds back to the application is called a *response step*.

Given any algorithm A, a *run* of A is an infinite sequence of steps of A such that the following properties hold for each process p: (1) initially, $mset = \emptyset$, (2) the current state in the first step of p is a special state *Init*, (3) for each step $< p, M, * >$, and for every message $m \in M$, p is the receiver of m and *mset* contains m immediately before the step $< p, M, * >$ is taken, and (4) if there is a step that puts a message m in *mset* such that p is the receiver of m and p takes an infinite number of steps, then there is a subsequent step $< p, M, * >$ such that $m \in M$.

A *partial run* is a finite prefix of some run. A (partial) run *rn extends* some partial run *pr* if *pr* is a prefix of *rn*. At the end of a partial run, all messages that are sent but not yet received are said to be *in transit*. We say that a process is *correct* in a run if it takes an infinite number of steps in that run. Otherwise the process is *faulty*. In a run of our model, any number of readers or the writer may be faulty, and at most $t \leq S$ servers might be faulty. We say that a (faulty) process p *crashes* at step *sp* in a run, if *sp* is the last step of p in that run.

2.2 Atomic Register

A sequential register is a data structure accessed by a single process. It provides two operations: write(v), which stores v in the register and returns OK, and read(), which returns the last value stored. An atomic register is a distributed data structure that may be concurrently accessed by multiple processes and yet provides an "illusion" of a sequential register to the accessing processes. We refer the readers to [4–6, 9] for a formal definition of an atomic register, and we simply recall below what is required to state and prove our results.

We assume that each process is asked by its application to invoke at most one operation at a time (i.e., a process does not take the next invocation step until it takes the response step for the current operation). Only readers invoke read on the register and only the writer invokes write on the register. We further assume that the initial value of a register is a special value $\perp$, which is not a valid input value for a write. In a run, an *operation* consists of an invocation step and (possibly) a matching response step. We say that an operation is *incomplete* in a (partial) run if the (partial) run does not contain a matching response step for the invocation step of that operation; otherwise, the operation is *complete*. In any run, we say that an operation $op1$ *precedes* operation $op2$ (or $op2$ *succeeds* $op1$) if the response step of $op1$ precedes the invocation step of $op2$ in that run. If neither $op1$ nor $op2$ precedes the other, the operations are said to be *concurrent*.

An algorithm *implements* a register if every run of the algorithm satisfies *termination* and *atomicity* properties. Termination states that every operation invoked by a correct process completes. Below, we recall the definition of atomicity as given by Lemma 13.16 of [9].

A *history* of a partial run is a sequence of invocation and response steps of read or write operations in the same order as they appear in the partial run. The definitions of incomplete, preceding, succeeding and concurrent operations in a history are similar to those for in a run. We say that a history $H1$ *completes* history $H2$ if $H1$ can be obtained through the following modification of $H2$: for each invocation step sp of an incomplete operation in $H2$, either sp is removed from $H2$, or any valid matching response for that invocation step is appended to the end of $H2$.

A run satisfies atomicity, if for every history H' of any of its partial runs, there is a history H that completes H' and H satisfies the properties 1-3 below (Lemma 13.16 of [9]). Let Π be the set of all operations in H. There is an irreflexive partial ordering $\prec$ of all the operations in H such that: (1) if $op1$ precedes $op2$ in H then it is not the case that $op2 \prec op1$, (2) if $op1$ is a write operation in Π and $op2$ is any other operation in Π, then either $op1 \prec op2$ or $op2 \prec op1$ in Π, and (3) the value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write operation).

For the *single-writer* registers, we can simplify the above definition of atomicity. In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. We assume without loss of generality that every write operation writes an unique value. Denote by wr_k the k^{th} write in a run ($k \geq 1$), and by val_k the value written by the k^{th} write. Let $val_0 = \perp$. We say that a partial run satisfies atomicity if the following properties hold: (1) if a read returns x then there is k such that $val_k = x$, (2) if a read rd is complete and it succeeds some write wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$, (3) if a read rd returns val_k ($k \geq 1$), then wr_k either precedes rd or is concurrent with rd, and (4) if some read $rd1$ returns val_k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns val_l, then $l \geq k$.

2.3 Fast Implementations

Basically, we say that a read or a write operation is *fast* if it completes in one communication round-trip. In other words, in a fast read, (1) the reader sends messages to a subset of processes in the system (possibly all processes), (2) processes on receiving such a message reply to the reader before receiving any other messages,[5] and (3) the reader on receiving a

[5]More precisely, any process p on receiving a message m in step $sp1 = < p, M, * >$ ($m \in M$), where m is sent by a reader on invoking a read, replies to m either in step $sp1$ itself, or in a subsequent step $sp2$,

```
at the writer w
procedure initialization:
    ts ← 1, rCounter ← 0
procedure write(v)
    send(WRITE, ts, rCounter) to all servers
    wait until receive(WRITEACK, ts, ∗, rCounter) from S − t servers
    ts ← ts + 1
    return(OK)

at each reader r_i
procedure initialization:
    ts ← 0; rCounter ← 0; maxTS ← 0
procedure read()
    rCounter ← rCounter + 1; ts ← maxTS
    send(READ, ts, rCounter) to all servers
    wait until receive(READACK, ∗, ∗, rCounter) from S − t servers
    rcvMsg ← {m|r_i received m = (READACK, ∗, ∗, rCounter)}
    maxTS ← Maximum{ts′| (READACK, ts′, ∗, rCounter) ∈ rcvMsg}
    maxTSmsg ← {m|m.ts = maxTS and m ∈ rcvMsg}
    if there is a ∈ [1, R + 1] and there is MS ⊆ maxTSmsg s.t., (|MS| ≥ S − at) and (| ∩_{m∈MS} m.seen| ≥ a) then
        return(maxTS)
    else
        return(maxTS − 1)

at each server s_i
procedure initialization:
    ts ← 0; seen ← ∅; counter[0...R] ← [0...0]
upon receive(msgType, ts′, rCounter′) from q ∈ {w, r_1, ..., r_R} and rCounter′ ≥ counter[pid(q)] do
    if ts′ > ts then
        ts ← ts′; seen ← {q}
    else
        seen ← seen ∪ {q}
    counter[pid(q)] ← rCounter′
    if msgType = READ then
        send(READACK, ts, seen, rCounter′) to q
    else
        send(WRITEACK, ts, seen, rCounter′) to q
```

Figure 1: Fast SWMR atomic register implementation with $R < \frac{S}{t} - 2$

sufficient number of such replies returns from the read. Recall that implementations need to tolerate the crash of the writer, any reader, and up to t servers. Hence, in order to ensure termination, the reader cannot wait for replies from any other reader, or writer, or more than $S - t$ servers. We similarly say that a write operation is fast if it completes in one round-trip.

We say that an implementation has fast reads (or writes) if every complete read (resp. complete write) operation in every run is fast. A *fast implementation* is an implementation in which both reads and writes are fast. For an implementation that has fast reads, we can say without ambiguity that the messages sent by a reader, on invoking a read, are of type READ, and the messages sent by a process to the reader, on receiving a READ message, of type READACK. Similarly, we define WRITE and WRITEACK messages for fast writes.

such that p does not receive any message in any step between $sp1$ and $sp2$ (including $sp2$). Intuitively, this requirement forbids the processes to wait for some other message before replying to m.

3 A Fast Implementation

We describe in this section a fast implementation assuming $R < \frac{S}{t} - 2$ (Figure 1). For simplicity of presentation, we assume that the writer writes timestamps, and the readers read back timestamps. We ignore the value associated with the timestamp for now. Later we explain how to trivially modify our algorithm such that the writer and the readers associate some value with a timestamp.

The procedure for write is similar to that in [1]. On invoking a write, the writer increments its timestamp and sends a WRITE message with the timestamp to all servers. Servers on receiving the message store the timestamp, and send WRITEACK messages back to the writer. The writer returns OK once it has received WRITEACK messages from $S - t$ servers.

Implementing a fast read is more involved. Recall that, to maintain atomicity, a read needs to return a timestamp that is not lower than the timestamp of the last completed write, and has to guarantee that no subsequent read returns a lower timestamp. Our read procedure collects timestamps from $S - t$ servers (by sending READ messages and receiving READACK messages from the servers), and selects the highest timestamp, denoted by $maxTS$ in Figure 1. Then the reader checks if $maxTS$ has been seen by a "sufficient" number of servers and readers. If so, the read returns $maxTS$, else it returns $maxTS - 1$. The heart of the algorithm is the predicate for checking whether the latest value has been seen by a sufficient number of processes: (1) the predicate is true whenever the write with timestamp $maxTS$ precedes the current read, and (2) if there is no write with a timestamp higher than $maxTS$, then if the predicate is true for the current read, it is also true for all subsequent reads. In order to construct such a predicate however, the servers need to record more information than just the latest timestamp, as we explain below.

Consider the case of a write with timestamp ts that is followed by a read. In the first partial run pr_1, the write completes by writing ts at $S - t$ servers, say the set of servers be S_1. Subsequently, a reader reads from a set S_2 (of $S - t$ servers) that overlaps at $S - 2t$ servers with S_1, i.e., misses t servers in S_1. By atomicity, the read returns ts. In the second partial run pr_2, the write is incomplete and the writer writes ts only to $S - 2t$ servers in $S1 \cap S2$. A subsequent reader that reads from S_2 cannot distinguish pr_1 from pr_2, and returns ts. If we extend each partial run with another read by a distinct reader that misses t servers from $S1 \cap S2$, it is easy to see that the new read has to return ts, even if it sees ts at $S - 3t$ servers that have already replied to both the write and the first read. Thus, we see that any reasonable predicate for fast reads must depend on the number of servers, as well as the number of readers, that have seen the most recent timestamp. Since any number of readers might crash, a reader cannot wait for the replies from other readers, but rather indirectly collect information about other readers from the servers.

Generalizing the above argument gives us the desired predicate. Along with the latest timestamp ts, every server maintains the list of readers and writer to which the server has replied after updating its timestamp to ts (including the reader or the writer which updated the timestamp of the server to ts). This set is denoted by *seen* in Figure 1. The predicate for the read procedure is as follows: if there is $a \geq 1$ such that the reader receives $maxTS$ in at least $S - at$ messages, and there are at least a processes that are in the list *seen* of each of those $S - at$ messages, then the predicate is true.

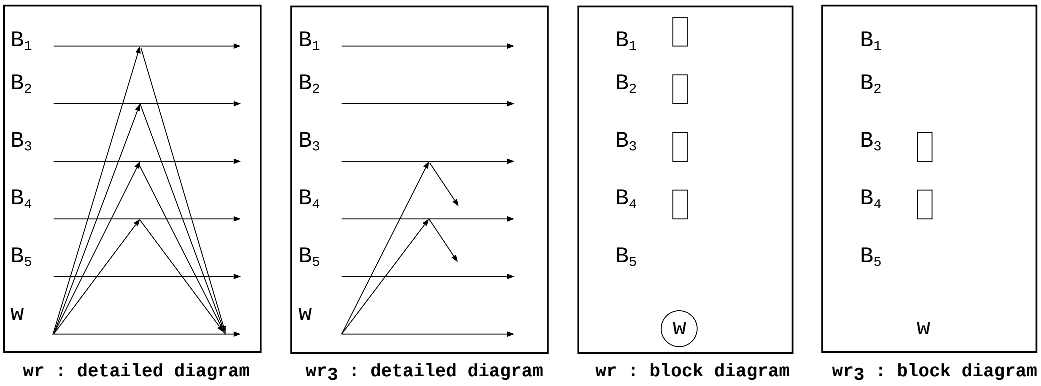

Figure 2: Partial writes: wr_i

In addition, every reader r_i maintains a variable $rCounter$ that counts the number of reads of r_i. The servers keep an array, $counter$, such that $counter[i]$ contains the latest value of $rCounter$ that the server has received from r_i.[6] This helps distinguish READ and READACK messages from different reads of the same reader. At the writer, the variable $rCounter$ is always 0; the messages from different writes are distinguished by their respective timestamps.

This completes the brief description of the register implementation. (The proof of correctness is presented in Appendix A.) We now describe how to modify the algorithm so as to associate values with timestamps. In the modified algorithm, in each write, the writer attaches two tags with the timestamp, containing the current value to be written and the value of the immediately preceding write. (The value of the immediately preceding write is $\perp$ for the first write operation.) If the reader returns $maxTS$ in the original algorithm, then it returns the current value attached to $maxTS$ in the modified algorithm. If the reader returns $maxTS - 1$ in the original algorithm, it returns the other tag attached to $maxTS$ in the modified algorithm.

4 Optimality

The following proposition states that the resilience required by our fast implementation is indeed necessary.

Proposition 1. *Let $t \geq 1$ and $R \geq 2$. If $R \geq \frac{S}{t} - 2$, then there is no fast atomic register implementation.*[7]

Preliminaries. Suppose by contradiction that $R \geq \frac{S}{t} - 2$ and there is a fast implementation I of an atomic register. Given that $t \geq S/(R+2)$, we can partition the set of servers into $R+2$ subsets (which we call *blocks*), denoted by B_i ($1 \leq i \leq R+2$), and each of size less than or equal to t.[8]

[6] In the algorithm, $pid(q)$ is a function that maps the writer w to 0, and every reader r_i to i.

[7] If $t = 0$ then it is easy to design a fast implementation irrespective of the number of readers. If $R = 1$, then a fast implementation is possible whenever $t < S/2$.

[8] For instance, one such partition is: for $1 \leq i \leq R+1$, $B_i = \{s_j \mid (\lfloor \frac{S}{R+2} \rfloor (i-1)+1) \leq j \leq (\lfloor \frac{S}{R+2} \rfloor i)\}$, and

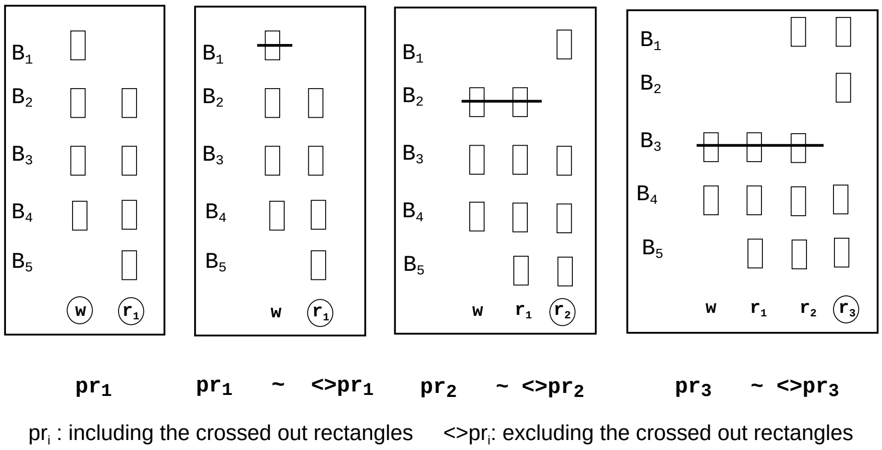

Figure 3: Partial runs: pr_i and $\triangle pr_i$

Since the writer, any number of readers, and up to t servers might crash in our model, the invoking process can only wait for reply messages from $S-t$ servers. Given that we assume a fast implementation, on receiving a READ (or a WRITE) message, the servers cannot wait for messages from other processes, before replying to the READ (or the WRITE) message. We can thus construct partial runs of a fast implementation such that only the READ (or the WRITE) messages from the invoking processes to the servers, and the replies from the servers to the invoking processes, are delivered. All other messages remain in transit. In particular, no server receives any message from other servers, and no invoking process receives any message from other invoking processes. In our proof, we only construct such partial runs.

We say that an *incomplete operation op skips* a set of blocks BS in a partial run, where $BS \subseteq \{B_1, ..., B_{R+2}\}$, if (1) no server in any block $B_i \in BS$ receives any READ or WRITE message from *op* in that partial run, (2) all other servers receive the READ or the WRITE message from *op* and reply to that message, and (3) *all those reply messages are in transit.* We say that a *complete operation op skips* a block B_i in a partial run, if (1) no server in B_i receives any READ or WRITE message from *op* in that partial run, (2) all servers that are not in B_i receive the READ or WRITE message from *op* and reply to that message, and (3) the invoking process *receives all those reply messages and completes the operation.*

Proof. To derive a contradiction, we construct a partial run of the fast implementation I that violates atomicity: a partial run in which some read returns 1 and a subsequent read returns an older value, namely, the initial value of the register, $\perp$.

Partial writes. Consider a partial run wr in which w completes write(1) on the register. The operation skips B_{R+2}. We define a series of partial runs each of which can be extended to wr. Let wr_{R+2} be the partial run in which w has invoked the write and has sent the

$B_{R+2} = \{s_j \mid (\lfloor \frac{S}{R+2} \rfloor (R+1)) \leq j \leq S\}$. However, if $R > S-2$ then the above partitioning is not possible. In that case we consider a system where the number of readers is $S-2$ and the set *readers* is $\{r_1, ..., r_{S-2}\}$, and then show the impossibility. The impossibility still holds if we add more readers to this system (i.e., $R > S-2$).

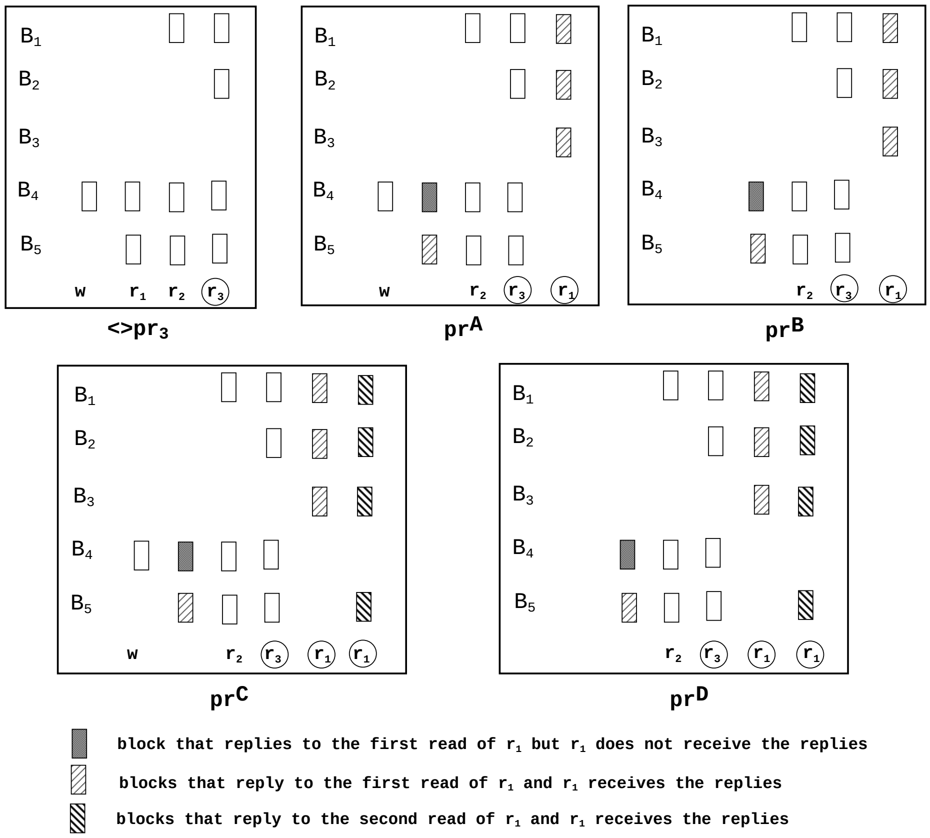

Figure 4: Partial runs: pr^A, pr^B, pr^C and pr^D

WRITE message to all processes, and all WRITE messages are in transit. For $1 \leq i \leq R+1$, we define wr_i as the partial run which contains an incomplete write(1) operation that skips $\{B_{R+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. We make the following observations: (1) for $1 \leq i \leq R+1$, wr_i and wr_{i+1} differ only at servers in B_i, (2) wr is an extension of wr_1, such that, in wr, w receives the replies (that are in transit in wr_1) and completes the write operation, and hence, (3) wr and wr_1 differ only at w.

Block diagrams. We illustrate a particular instance of the proof through Figure 2, Figure 3 and Figure 4, where $R = 3$ and the set of servers are partitioned into five blocks, B_1 to B_5. We depict a (possibly incomplete) operation *op* through a set of rectangles, (generally) arranged in a single column, with the name of the invoking process at the bottom of the column. If the operation is complete, we circle the name of the invoking process. In the column corresponding to some operation *op*, we draw a rectangle in the i^{th} row, if all servers in block B_i have received the READ or WRITE message from *op* and have sent the reply messages, i.e., we draw a rectangle in the i^{th} row if *op* does not skip B_i. (For example, Figure 2 presents detailed diagrams of two partial writes with corresponding block diagrams.)

Appending reads. Partial run pr_1 extends wr by appending a complete read by r_1 that skips block B_1. By atomicity, the read returns 1. Observe that r_1 cannot distinguish pr_1

from some partial run $\triangle pr_1$, where $\triangle pr_1$ is an extension of wr_2 obtained by appending a complete read by r_1 that skips B_1. To see why, notice that wr and wr_2 differ at w and at block B_1, and r_1 does not receive any message from those processes in both runs. Thus r_1's read returns 1 in $\triangle pr_1$.

Starting from $\triangle pr_1$, we iteratively define the following partial runs for $2 \leq i \leq R$. Partial run pr_i extends $\triangle pr_{i-1}$ by appending a complete read by r_i that skips B_i. Partial run $\triangle pr_i$ is constructed by deleting from pr_i, all steps of the servers in block B_i. Since the last read in pr_i by reader r_i skips block B_i, r_i cannot distinguish pr_i from $\triangle pr_i$. More precisely, partial run $\triangle pr_i$ extends wr_{i+1} by appending the following i reads one after the other:[9] for $1 \leq h \leq i$, r_h does a read that skips $\{B_j | h \leq j \leq i\}$. Figure 3 depicts block diagrams of pr_i and $\triangle pr_i$ with $R = 3$. (The deletion of steps to obtain $\triangle pr_i$ from pr_i is shown by crossing out the rectangles corresponding to the deleted steps.)

Reader r_1's read in $\triangle pr_1$ returns 1. Since pr_2 extends $\triangle pr_1$, by atomicity, r_2's read in pr_2 returns 1. However, as r_2 cannot distinguish pr_2 from $\triangle pr_2$, r_2's read in $\triangle pr_2$ returns 1. In general, since pr_i extends $\triangle pr_{i-1}$, and r_i cannot distinguish pr_i from $\triangle pr_i$ (for all i such that $2 \leq i \leq R$), it follows from a trivial induction that r_i's read in $\triangle pr_i$ returns 1. In particular, r_R reads 1 in $\triangle pr_R$.

Partial run pr^A. Consider the partial run $\triangle pr_R$: wr_{R+1} extended by appending R reads by each reader r_h ($1 \leq h \leq R$) such that r_h's read skips $\{B_j | h \leq j \leq R\}$. Observe that the read by r_1 is incomplete in $\triangle pr_R$: only servers in B_{R+1} and B_{R+2} send replies to r_1, and those reply messages are in transit. Furthermore, in $\triangle pr_R$, only the servers in B_{R+1} receive the WRITE message from the write(1) operation. Consider the following partial run pr^A which extends $\triangle pr_R$ as follows. After $\triangle pr_R$, (1) r_1 receives the replies of its READ messages from B_{R+2} (that were in transit in $\triangle pr_R$), (2) the servers in B_1 to B_R receive the READ message from r_1 (that were in transit in $\triangle pr_R$) and reply to r_1, (3) reader r_1 receives those replies from servers in B_1 to B_R, and then r_1 returns from the read operation. (Notice that, r_1 received replies from $R+1$ blocks, and so, must return from the read.) However, r_1 does not receive the replies from servers in B_{R+1} (that were in transit in $\triangle pr_R$). Figure 4 depicts block diagrams for pr^A with $R = 3$.

Partial run pr^B. Consider another partial run pr^B with the same communication pattern as pr^A, except that write(1) is not invoked at all, and hence, servers in B_{R+1} do not receive any WRITE message (Figure 4). Clearly, only servers in B_{R+1}, the writer, and the readers from r_2 to r_R can distinguish pr^A from pr^B. Reader r_1 cannot distinguish the two partial runs because it does not receive any message from the servers in B_{R+1}, the writer, or the other readers. By atomicity, r_1's read returns (the initial value of the register) $\perp$ in pr^B because there is no write($*$) operation in pr^B, and hence, r_1's read returns $\perp$ in pr^A as well.

Partial runs pr^C and pr^D. Notice that, in pr^A, even though r_1's read returns $\perp$ after r_R's read returns 1, pr^A does not violate atomicity, because the two reads are concurrent. We construct two more partial runs: (1) pr^C is constructed by extending pr^A with another complete read by r_1, which skips B_{R+1}, and (2) pr^D is constructed by extending pr^B with

[9] The first $i-1$ reads are incomplete whereas the last one is complete.

another complete read by r_1, which skips B_{R+1} (Figure 4). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips B_{R+1} (i.e., the servers which can distinguish pr^A from pr^B), it follows that r_1 cannot distinguish pr^C from pr^D as well. Since there is no write operation in pr^D, r_1's second read returns $\perp$ in pr^D, and hence, r_1's second read in pr^C returns $\perp$. Since pr^C is an extension of pr^A, r_R's read in pr^C returns 1. Thus, in pr^C, r_1's second read returns $\perp$ and succeeds r_R's read which returns 1. Clearly, partial run pr^C violates atomicity. □

5 MWMR Implementations

The existence of a fast implementation of SWMR register when $R < \frac{S}{t} - 2$ raises the question of the possibility of such an implementation for the multi-writer case. In the following proposition we show that if any number of readers and writers may fail, no fast implementation of MWMR registers can tolerate the failure of even one server.

The definition of atomicity from Lemma 13.16 of [9], recalled in Section 2.2, is valid for MWMR registers as well. We define a fast implementation of a MWMR register as in the SWMR case: in every run, every complete operation takes at most one round-trip to complete.

In the proof below, we use three simple properties of MWMR registers which can be easily deduced from the atomicity property. In any partial run: (property **P1**) if a write wr that writes v, precedes some complete read rd, and all other writes precede wr, then rd returns v, (property **P2**) if there are two reads such that all writes precede both reads, then the reads do not return different values, and (property **P3**) if all writes are complete and precede some complete read rd, then rd returns the value written by some complete write. In the following, we denote by W the number of writers in the system.

Proposition 2. *Let $W \geq 2$ and $R \geq 2$. If $t \geq 1$, then there is no fast atomic register implementation.*

Proof. It is sufficient to show the impossibility in a system where $W = R = 2$, and $t = 1$. We assume that any number of readers and writers may fail. Let the writers be w_1 and w_2, and the readers be r_1 and r_2. Let s_1 to s_S be the servers. Suppose by contradiction that there is a fast implementation of an atomic register in this system. To show the desired contradiction, we construct a series of runs, each consisting of two writes followed by a read.

Since the writer, any number of readers, and up to t servers might crash in our model, the invoking process can only wait for reply messages from $S - t$ servers. Given that we assume a fast implementation, on receiving a READ (or a WRITE) message, the servers cannot wait for messages from other processes, before replying to the READ (or the WRITE) message. We can thus construct partial runs of a fast implementation such that only READ (or WRITE) messages from the invoking processes to the servers, and the replies from the servers to the invoking processes, are delivered. All other messages remain in transit. In particular, no server receives any message from other servers, and no invoking process receives any message from other invoking processes. In our proof, we only construct such partial runs.

We say that a complete operation *op skips* a server s_i in a partial run if every server distinct from s_i receives the READ or the WRITE message from *op* and replies to that message, *op* receives those replies and returns, and all other messages are in transit. In other words, only s_i does not receive any READ or WRITE message from *op*. Since $t = 1$, any complete operation may skip at most one server. If a complete operation does not skip any servers, we say that the operation is *skip-free*.

Consider a partial run run_1 constructed with the following three non-concurrent complete operations: (a) a skip-free write(2) by w_2, that precedes (b) a skip-free write(1) by w_1, that in turn precedes (c) a skip-free read() by r_1. From property P1, the read returns 1.

We now construct a similar partial run run_2 in which the order of the two writes are interchanged: (a) a skip-free write(1) by w_1, that precedes (b) a skip-free write(2) by w_2, that in turn precedes (c) a skip-free read() by r_1. From property P1, the read returns 2.

Consider a series of partial runs run^i, where i varies from 1 to $S+1$. We define run^1 to be run_1. We iteratively define the remaining partial runs. We define run^{i+1} to be identical to run^i except in the following: s_i receives the WRITE message (and replies to that message) from w_1 before the message from w_2 (i.e., s_i receives the WRITE messages in the opposite order in run^{i+1} from that in run^1). Since servers do not receive any messages from other servers in the partial runs we construct, the only server that can distinguish run^i from run^{i+1} is s_i. Also w_1, w_2 and r_1 can distinguish the two partial runs. It is easy to see that no server can distinguish run^{S+1} from run_2, because in both runs the WRITE messages by every server is received in the opposite order from that in run^1. Hence, r_1 cannot distinguish the two runs as well. Thus r_1 returns 2 in run^{S+1}, and run^{S+1} and run_2 differ only at $w1$ and $w2$. Since r_1 returns 1 in run^1, 2 in run^{S+1}, and from property P3, either 1 or 2 in run^i $(2 \leq i \leq S)$, there are two partial runs, run^{i1} and run^{i1+1}, such that $1 \leq i1 \leq S$ and the read by r_1 returns 1 in run^{i1} and returns 2 in run^{i1+1}.

Consider a partial run run' which extends run^{i1} with a read by r_2 that skips s_{i1}. From property P2, it follows that r_2 returns 1. We similarly construct a partial run run'' which extends run^{i1+1} with a read by r_2 that skips s_{i1}. Recall that only w_1, w_2, r_1 and s_{i1} can distinguish run^{i1} from run^{i1+1}. Since r_2 skips s_{i1} in both run' and run'', r_2 cannot distinguish the two partial runs. Thus r_2 returns 1 in run''. However, r_1 returns 2 in run^{i1+1}, and hence, it returns 2 in run'' as well. Clearly, run'' violates property P2. $\square$

To see why the above proof does not apply to the single writer case, observe that in most partial runs used in the proof, the two writes are concurrent. However, in our system model, a process can invoke at most one operation at a time. Thus we cannot construct partial runs with concurrent writes in the single-writer case.

6 Concluding Remarks

We say than an implementation is *semi-fast* if the implementation has fast reads *or* fast writes (not necessarily both). Semi-fast implementations might be interesting in systems where one of the operations (say read) is more frequent than the other. It would be interesting to investigate, given S and t, whether we can increase the number of readers, if we consider semi-fast implementations.

Another direction for further investigation could be designing hybrid implementations, i.e., implementations that tolerate failures of any minority of servers, which (from Proposition 1) cannot be fast, but that would complete their operations in a fast mode whenever a sufficient number of servers are available. Hybrid implementations might be useful in a dynamic setting such as that of [8]: depending on the current configuration, such implementations would be able to switch between fast and slow modes.

Acknowledgment. We thank Petr Kouznetsov and Eric Ruppert for their helpful comments.

References

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

[2] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[3] R. Fan and N. Lynch. Efficient replication of large data objects. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC-17)*, Sorrento, Italy, October 2003.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[5] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[6] L. Lamport. On interprocess communication – parts I-II. *Distributed Computing*, 1(2):77–101, May 1986.

[7] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, Seattle, Washington, June 1997. IEEE.

[8] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC-16)*, Toulouse, France, October 2002.

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

A Correctness of the fast implementation

We prove here the correctness of the fast implementation of Figure 1. We do not assume that the lines in Figure 1 are atomic: processes may crash in the middle of a line or in between two lines. In particular, while sending messages to a set of processes, the sending process may crash after sending messages to an arbitrary subset.

It is obvious that the read and the write procedures complete in one round-trip. To show atomicity, we recall that the write procedure directly writes the timestamp. Thus the conditions for atomicity of the single writer case in Section 2.2 reduce to the following: (1) if a read returns, it returns a non-negative integer, (2) if a read rd is complete and it succeeds some write(k), then rd returns l such that $l \geq k$, (3) if a read rd returns k ($k \geq 1$), then write(k) either precedes rd or is concurrent with rd, and (4) if some read $rd1$ returns k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns l, then $l \geq k$. The proofs of the first and the third conditions are trivial. Below, we show the other two. In the proofs we refer to the global clock; however processes do not have access to this clock.

Lemma 3. *If a server sets ts to x at time T, then the server never sets ts to a value that is lower than x after time T.*
Proof. Obvious from line 27. □

Lemma 4. *If a read() sends* READ *messages with $ts = x$, then the read does not return a value smaller than x.*
Proof. Suppose read rd by r_i sends a READ message with $ts = x$. From line 27, every READACK message received by rd is with $ts \geq x$. Let z be the maximum timestamp received by rd (i.e., $maxTS$ computed in line 17). Notice that rd returns either z or $z - 1$. There are the following two cases to consider. (1) If $z > x$, then clearly, the return value is not smaller than x. (2) If $z = x$, then every READACK message received by rd has $ts = x$ and has $r_i \in seen$. Since rd receives $S - t$ READACK messages, the predicate in line 19 of rd holds with $a = 1$. Hence, rd returns x. □

Lemma 5. *If a read rd is complete and it succeeds some write(k), then rd returns l such that $l \geq k$.*
Proof. Suppose that write wr (by w) writes k and precedes read rd (by reader r_j). Let $S1$ be the set of $S - t$ servers from which wr received WRITEACK messages in line 6, and let $S2$ be the set of $S - t$ servers from which rd received READACK messages in line 15. Let $S12 = S1 \cap S2$. Obviously, $|S12| \geq S - 2t$. Let z be the maximum timestamp received by rd from servers in $S2$. Observe that rd returns either z or $z - 1$.

When a server in $S1$ replies to a WRITE message from wr, its ts is k.[10] From Lemma 3, servers in $S1$ (and hence, in $S12$) reply $ts \geq k$ to rd because wr precedes rd). Thus, the maximum timestamp z received by rd is greater than or equal to k. There are the following two cases to consider:

1. $z > k$: As rd returns either z or $z - 1$, it follows that rd does not return a timestamp lower than k.

[10]The server's ts is not higher than k because, unless the writer receives WRITEACK from all servers in $S1$, it does not complete write(k), and hence, no timestamp higher than k is present in the system until all servers in $S1$ reply to write(k).

2. $z = k$: We know that every server in $S12$ replies to rd with $ts \geq k$, and $z = k$ is the maximum timestamp received by rd from servers in $S2 \supseteq S12$. Thus every server in $S12$ replies $ts = k$ to rd. Let MS be the set of READACK messages sent by servers in $S12$ to rd. Since every server in $S12$ replies $ts = k$ to wr before sending $ts = k$ to rd, for every message m in MS, $w \in m.seen$. Furthermore, from line 30, every server includes r_j in its $seen$ before sending READACK message to r_j. Thus for every m in MS, $r_j \in m.seen$. Thus, $\{w, r_j\} \subseteq \cap_{m \in MS} m.seen$. As $|S12| \geq S - 2t$, it follows that, in rd, the predicate in line 19 holds with $a = 2$. Consequently, rd returns $z = k$. □

Lemma 6. *If some read $rd1$ returns x ($x \geq 0$) and a read $rd2$ that succeeds $rd1$ returns y, then $y \geq x$.*

Proof. Suppose that read $rd1$ by process r_j returns x, read $rd2$ by process r_k returns y, and $rd1$ precedes $rd2$. Suppose $r_j = r_k$. Then, in the read immediately after $rd1$, r_j sends a READ message with $ts \geq x$, and hence, from Lemma 4, the read returns a value greater than or equal to x. Using Lemma 4 and a simple induction, we can derive that any read by r_j which follows $rd1$ (including $rd2$) returns $ts \geq x$. So in the rest of the proof we assume that $r_j \neq r_k$.

Let $S1$ and $S2$ be the set of servers (of size $S - t$) from which reads $rd1$ and $rd2$, respectively, received $S - t$ READACK messages in line 15. Let $TS1$ be the highest timestamp received by $rd1$ from processes in $S1$ (i.e., the $maxTS$ evaluated in line 17 of $rd1$). Similarly, let $TS2$ be the highest timestamp received by $rd2$ from the processes in $S2$. There are the following two cases to consider: (1) the predicate in line 17 does not hold in $rd1$, and (2) the predicate in line 17 holds in $rd1$. For both cases, we show that $y \geq x$.

Case 1: It follows that $x = TS1 - 1$. Thus some server has sent $ts = TS1 = x + 1$ to $rd1$, and hence, write($x + 1$) has started before $rd1$ is completed. Thus write(x) has completed before $rd1$ is completed. Since $rd1$ precedes $rd2$, it follows that write(x) precedes $rd2$. From Lemma 5, $rd2$ returns $y \geq x$.

Case 2: It follows that $x = TS1$, and there is some $a \in [1, R+1]$ such that there is a set MS consisting of at least $S - at$ messages received by $rd1$ with $ts = x$ and $|\cap_{m \in MS} m.seen| \geq a$. Let $S12 \subseteq S1$ be the set of servers which sent the messages that are in MS. Since $a \in [1, R+1]$ and $t < S/(R+2)$, $|S12| = |MS| = S - at > t$. There are the following two cases to consider: (1) $y = TS2$, and (2) $y = TS2 - 1$. For both cases, we show that either $y \geq x$ or the case is impossible.

Case 2.1: $y = TS2$. Since, $|S12| > t$ and $|S2| = S - t$, there is a server $s_i \in S2 \cap S12$. Since $rd1$ precedes $rd2$, s_i first replies $ts = x$ to $rd1$ then replies to $rd2$. From Lemma 3, it follows that s_i replies to $rd2$ with $ts \geq x$. Thus the highest ts in $S2$ (i.e., $TS2 = y$) is greater than or equal to x.

Case 2.2: $y = TS2 - 1$. There are the following two subcases to consider: (1) $y + 1 \neq x$ and (2) $y + 1 = x$.

Case 2.2.1: $y+1 \neq x$: As in case 2.1, there is a server $s_i \in S2 \cap S12$, and s_i replies to $rd2$ with $ts \geq x$. Thus the highest ts in $S2$ (i.e., $TS = y+1$) is greater than or equal to x. Since $y+1 \neq x$, it follows that $y+1 > x$, and hence, $y \geq x$.

Case 2.2.2: $y+1 = x$: Consider the set of servers $S2 \cap S12$. As $|S12| = S - at$ and $|S2| = S - t$, so $|S2 \cap S12| \geq S - (a+1)t \geq 1$. Since $rd1$ precedes $rd2$ and processes in $S12$ replies $ts = x$ to $rd1$, processes in $S2 \cap S12$ reply to $rd2$ with $ts \geq x$. Since $y+1$ is the maximum ts in $S2$, every process in $S2 \cap S12$ replies to $rd2$ with $ts = x = y+1$. There are the following two cases to consider: (1) $a \leq R$ and (2) $a = R+1$.

Case 2.2.2.1: $a \leq R$. Then $|S2 \cap S12| \geq S - (a+1)t > t$. Let $MS1$ be the set of READACK messages from processes in $S2 \cap S12$ to $rd1$. From the definition of $MS1$ and MS, $MS1 \subseteq MS$.[11] Thus, $\cap_{m \in MS1} m.seen \supseteq \cap_{m \in MS} m.seen$. Thus, $|\cap_{m \in MS1} m.seen| \geq a$. There are two cases to consider: (1) $r_k \notin \cap_{m \in MS1} m.seen$ and (2) $r_k \in \cap_{m \in MS1} m.seen$.

Case 2.2.2.1.1: $r_k \notin \cap_{m \in MS1} m.seen$. Let $MS2$ be the set of messages received by $rd2$ from processes in $S2 \cap S12$. For any server $s_i \in S2 \cap S12$, let $m1_i$ and $m2_i$ be the messages sent by s_i in $MS1$ and $MS2$ respectively. We know that $m1_i.ts = m2_i.ts = x$. Since $m1_i$ is sent before $m2_i$ and the ts is the same in both messages, $m1_i.seen \subseteq m2_i.seen$. Thus $\cap_{m \in MS1} m.seen \subseteq \cap_{m \in MS2} m.seen$. Since every process which replies to $rd2$, first adds r_k to its *seen* set, $r_k \in \cap_{m \in MS2} m.seen$. Since $r_k \notin \cap_{m \in MS1} m.seen$, it follows that $|\cap_{m \in MS2} m.seen| \geq |\cap_{m \in MS1} m.seen| + 1 \geq a+1$. Since $|S2 \cap S12| \geq S - (a+1)t$, the number of message in $MS2$ is at least $S - (a+1)t$. As $a+1 \leq R+1$, the predicate in line 19 in $rd2$ holds with $a+1$. Thus, the timestamp returned by $rd2$ is $x = y+1$, a contradiction.

Case 2.2.2.1.2: $r_k \in \cap_{m \in MS1} m.seen$. Thus each server s_i in $S2 \cap S12$ has sent at least one READACK message with $ts = x$ to r_k, before s_i sent the $MS1$ message to r_j. Since the messages in $MS1$ are sent before the completion of $rd1$ (and hence, before the invocation of $rd2$), r_k has invoked at least one read before $rd2$. Let $rd2a$ be the last read of r_k which precedes $rd2$. Since $|S2 \cap S12| \geq S - (a+1)t > t$, there is at least one process s_i in $S2 \cap S12$ whose READACK message is received by $rd2a$, say message m. Now consider the last READACK message sent by s_i to r_k before $rd2$ is invoked, say message m'. Since we know that s_i sent a READACK message with $ts = x$ to r_k before sending a $MS1$ message (which was in turn sent before $rd2$ was invoked), from Lemma 3 it follows that m' was sent with $ts \geq x$. We now claim that $m = m'$. By definition of m', either $m = m'$ or m' is sent after m. Observe that s_i checks *counter*$[k]$ before replying to r_k. Thus, once m is sent by s_i, *counter*$[k]$ at s_i is set such that s_i can only reply to those message of r_k which are sent from $rd2a$ or a subsequent read of r_k. Thus, if m' is sent after m, then m' is sent in response to $rd2a$, or $rd2$, or a subsequent read of r_k. This contradicts the assumption that s_i replies only once to $rd2a$ (because channels do not duplicate messages) and m' is sent before $rd2$ is invoked. Thus $rd2a$ receives $m = m'$. We have already shown that m' is sent with $ts \geq x$. Hence the highest ts received by r_k in read $rd2a$ is greater than or equal to x. It follows that, in the sub-

[11] See Case 2 for the definition of MS.

sequent read $rd2$, reader r_k sends READ messages with $ts \geq x$. From Lemma 4, $rd2$ returns a timestamp greater than or equal to x. As $x = y+1$, $rd2$ does not return y: a contradiction.

Case 2.2.2.2: $a = R+1$. Since $|\{w, r_1, ..., r_R\}| = R+1$ and $|\cap_{m \in MS} m.seen| \geq a = R+1$, we have $r_k \in \cap_{m \in MS} m.seen$. Observe that $|S12| \geq S - at > t$. (Recall that $S12$ is the set of processes which sent the messages that are in MS.) Substituting $MS1$ by MS, and $S2 \cap S12$ by $S12$, in the argument for the previous case (Case 2.2.2.1.2), we can derive that $rd2$ returns a value greater than or equal to x: a contradiction. $\square$