How Fast can a Distributed Atomic Read be? [1]

PARTHA DUTTA, RACHID GUERRAOUI, RON R. LEVY and MARKO VUKOLIC
School of Computer & Communication Sciences, EPFL, Lausanne, Switzerland

We study efficient and robust implementations of an atomic read-write data structure over an asynchronous distributed message-passing system made of reader and writer processes, as well as failure prone server processes implementing the data structure. We determine the exact conditions under which every read operation involves *one-round* of communication with the servers. These conditions relate the number of readers to the tolerated number of faulty servers, in a general model with crash and arbitrary failures.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*; F.1.1 [**Computation by Abstract Devises**]: Models of Computation—*relations among models*

General Terms: Algorithms, Theory, Reliability

Additional Key Words and Phrases: atomic registers, shared-memory emulation, time-complexity

1. INTRODUCTION

Assigning a value to a variable and fetching a value from a variable are probably the most common instructions of any program. When several programs cooperate to achieve a common task, it is natural to provide them with means to perform those instructions through *shared* variables.

The *atomic* read-write data structure allows concurrent processes, each possibly running a different program, to share information through a common variable, as if they were accessing this variable in a sequential manner. This abstraction, usually called an *atomic register* [Lamport 1986] or simply a *register*, is fundamental in distributed computing and is at the heart of a large number of distributed algorithms [Herlihy 1991; Attiya and Welch 1998].

We study distributed implementations of this abstraction in a message passing system with no physical shared memory: a set of server processes provide the illusion to a set of reader and writer processes (clients) that the abstraction is a physical memory accessible to the clients.

We consider *robust* [Attiya et al. 1995], or *wait-free* [Herlihy 1991], implementations of this data structure where any read or write invocation by some client process eventually returns, independently of the operational status of other clients: all of them might have stopped their computation. The data structure itself is replicated over several servers to tolerate the failure of some of these servers.

Ensuring both atomicity and robustness is not trivial. Informally, atomicity requires that, even though each read or write operation may take an arbitrary period of time to complete, they appear to execute at some instant during their respective period of execution [Lamport 1986]. This requires ordering operations in a way that respects their physical order as well as the expected sequential specification of

[1] Elements of this paper appeared in a preliminary form in a paper with the same title in the proceedings of the ACM Symposium on Principles of Distributed Computing, 2004.

a read-write data structure: namely, a read should return the last written value.

To illustrate how an implementation can be robust yet achieve atomicity and motivate our quest for efficient implementations, consider the classical implementation from [Attiya et al. 1995] in the case of a single-writer multi-reader case, also called an SWMR register [Lamport 1986]. In [Attiya et al. 1995], readers and servers are the same set, the writer is one of the servers, and a minority of processes may fail by crashing, i.e., halting all their activities.

This implementation 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 servers. Every server, 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 servers, and selects the value v with the largest timestamp ts. Then the reader sends v and ts to all servers, and returns v on receiving acks from a majority of processes. Unlike the writer[2], a reader does not know the latest timestamp in the system, and hence, needs to spend one communication round-trip to discover the latest value, and then another round-trip to propagate the value to a majority of servers. The second round-trip is "required" because the latest value learned in the first round-trip might be present at only a minority of servers. 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.

It is easy to see how to reduce the time-complexity of a read 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. 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.

But can we do better? Is there a *fast* implementation where none of the operations (read or write) require more than one communication round-trip? This would clearly be optimal in terms of time-complexity.

With a single reader, it is easy to modify the algorithm of [Attiya et al. 1995] such that the read takes only one round-trip: the read can return the latest value learned from the servers in the first round trip, provided it is not older than the

[2]Given the 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.

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. 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.

Consider now the situation with another reader. 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 when $t < S/2$. But what if we tolerate fewer faulty servers?

We show in this paper that, interestingly, the existence of a fast SWMR implementation depends on the maximum number R of readers. We consider a general model where t among the set S of server processes on which the data structure is implemented can fail by crashing, or even deviate arbitrarily from their algorithm and be malicious: we denote by $b \leq t$ the number of malicious server failures.

We show that there is a fast implementation of a SWMR register *if and only if* the number of readers R is less than $\frac{S+b}{t+b} - 2$.

For simplicity of presentation, we first prove our result for the crash-stop case. ($R < \frac{S}{t} - 2$), and later generalize it to arbitrary failures (i.e., assuming $b \neq 0$). We first give an algorithm (i.e., a fast implementation) and then a lower bound (i.e., we prove the implementation is optimal).

—To get an intuition of our fast implementation, consider the algorithm sketched above [Attiya et al. 1995] and 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 atomicity 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 to return, we have every server maintain, besides the latest value, the set of readers to which the server has sent that value.

—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). By atomicity, 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 algorithm and lower bound are then extended, after a careful analysis of the impact of malicious servers, to the more general situation with arbitrary failures. We assume that, out of the t servers than can fail, up to $b \leq t$ processes can be malicious and we show that a fast implementation is possible *if and only if* the number of readers is less than $\frac{S+b}{t+b} - 2$.

—To complete the picture, we prove that it is impossible to have a *one-round* read algorithm with multiple writers [Lynch and Shvartsman 1997] (MWMR atomic register) even if only one server can fail and it can only do so by crashing.

The paper is organized as follows. Section 2 gives the system model and defines fast atomic implementations. We present a fast implementation assuming $R < \frac{S}{t} - 2$ in Section 4. We prove a tight bound for R in Section 5. Section 6 extends the previous results to the arbitrary failure model. Section 7 considers the multi-writer case. Section 8 discusses the impact of our results on the folklore theorem that every "atomic read must write". Section 9 summarizes the main results of the paper.

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 $\{w\}$,[3] and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel.

A distributed algorithm A is a collection of automata, where A_p is the automata assigned to process p. Computation proceeds in *steps* of A. A step of algorithm A is denoted by a pair of process id and a set of messages received in that step $< p, M >$ (M might be $\emptyset$). A *run* is an infinite sequence of steps of A. A *partial run* is a finite prefix of some run. A (partial) run r *extends* some partial run pr if pr is a prefix of r. At the end of a partial run, all messages that are sent but not yet received are said to be *in transit*. In any given run, any number of readers, the writer, and t out of S servers may crash.

2.2 Details of the System Model

The state of communication channels is viewed as a set of messages $mset$ 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 automata assigned to process p. Computation proceeds in *steps* of A. A step of A is denoted by a pair of process id and message set $< p, M >$ (M might be $\emptyset$). In step $< p, M >$, process p atomically does the following: (1) remove the messages in M from $mset$, (2) apply M and its current state st_p to A_p, which outputs a new state st'_p and a set of

[3]We discuss the multi-writer case in Section 7.

messages to be sent, and then (3) p adopts st'_p as its new state and puts the output messages in $mset$.

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. 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.

For presentation simplicity, we do not explicitly model the initial state of a process, nor the invocations and responses of operations. We assume that the algorithm A initializes the processes, and schedules invocation/response of operations (i.e., A modifies the states of the processes accordingly). However, we say that p invokes op at step sp, if A modifies the state of a process p in step sp so as to invoke an operation (and similarly for response).

3. ATOMICITY

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. An *incomplete* invocation step in a history is an invocation step without a matching response step in that history. We say that a history $H1$ *completes* history $H2$ if $H1$ can be obtained through the following modification of $H2$: for each incomplete invocation step sp in $H2$, either sp is removed from $H2$, or any valid matching response for that invocation is appended to the end of $H2$.

3.1 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 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 [Lamport 1986; Lynch 1996; Herlihy 1991; Herlihy and Wing 1990] 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 invokes at most one invocation at a time (i.e., does not invoke the next operation until it receives the response for the current operation). Only readers invoke reads on the register and only the writer invokes writes 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 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*. We say that an operation is

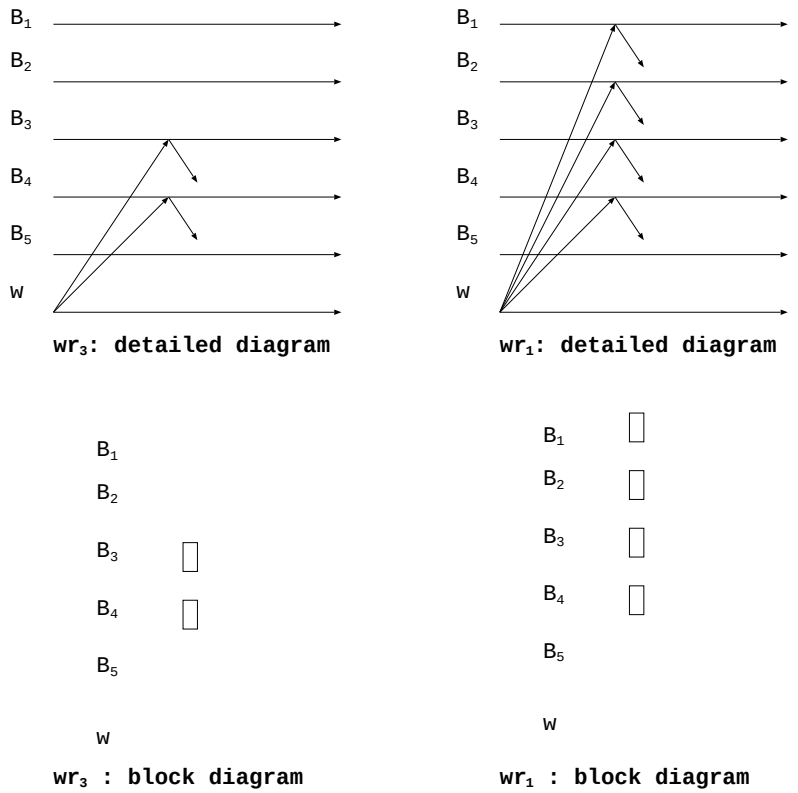


Fig. 1. Partial writes: wr_i

complete in a (partial) run if the run contains a response step for that operation.

An algorithm *implements* a register if every run of the algorithm satisfies *termination* and *atomicity* properties. Termination states that if a process invokes an operation, then eventually, unless that process crashes, the operation completes (even if all other client processes have crashed). Here we give a definition of atomicity for the *single-writer* registers.

In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. 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 to rd, and (4) if some read $rd1$ returns val_k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns val_l, then $l \geq k$.

3.2 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. 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 $sp2$, 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) the reader on receiving a 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.

4. A FAST IMPLEMENTATION

We describe in this section a fast implementation assuming $R < \frac{S}{t} - 2$ (the pseudo code of the implementation is given in Figure 2). 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 [Attiya et al. 1995]. 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 2. 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.

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

9: at each reader r_i
10: procedure initialization:
11:    ts ← 0; rCounter ← 0; maxTS ← 0
12: procedure read()
13:    rCounter ← rCounter + 1; ts ← maxTS
14:    send(READ, ts, rCounter) to all servers
15:    wait until receive(READACK, ∗, ∗, rCounter) from S − t servers
16:    rcvMsg ← {m|r_i received (READACK, ∗, ∗, rCounter)}
17:    maxTS ← Maximum{ts'| (READACK, ts', ∗, rCounter) ∈ rcvMsg}
18:    maxTSmsg ← {m|m.ts = maxTS and m ∈ rcvMsg}
19:    if there is a ∈ [1, R + 1] and there is MS ⊆ maxTSmsg s.t., (|MS| ≥ S − at) and
       (| ∩_{m∈MS} m.seen| ≥ a) then
20:       return(maxTS)
21:    else
22:       return(maxTS − 1)

23: at each server p_i
24: procedure initialization:
25:    ts ← 0; seen ← ∅; counter[0...R] ← [0...0]
26: upon receive(msgType, ts', rCounter') from q ∈ {w, r_1, ..., r_R} and rCounter' ≥
    counter[pid(q)] do
27:    if ts' > ts then
28:       ts ← ts'; seen ← {q}
29:    else
30:       seen ← seen ∪ {q}
31:    counter[pid(q)] ← rCounter'
32:    if msgType = READ then
33:       send(READACK, ts, seen, rCounter') to q
34:    else
35:       send(WRITEACK, ts, seen, rCounter') to q
```

Fig. 2. Fast SWMR atomic register implementation with $R < \frac{S}{t} - 2$

(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 2. 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 these $S-at$ messages, then the predicate is true.

In addition, every reader r_i maintains a variable $rCounter$ that counts the number of reads of r_i. The servers maintain an array, *counter*, such that $counter[i]$ contains the latest value of $rCounter$ that the server has received from r_i. In the algorithm, $pid(q)$ is a function that maps the writer w to 0, and every reader r_i to i. 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. 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. 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.

We now prove the correctness of the fast implementation in Figure 2. We do not assume that the lines in Figure 2 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. We assume that, if a process receives an incomplete message, the process can detect that the message is incomplete, and ignores such a message.

It is obvious that read and write procedures complete in one round-trip. To show atomicity, we recall that the write procedure directly writes the timestamp. Thus the conditions in Section 3.1 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 to rd.
(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 access this global clock.

LEMMA 1. *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 2. *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 $p_i \in seen$. Since rd receives $S - t$ READACK messages, the predicate in line 19 of rd holds with $a = 1$. Hence, rd returns x.

LEMMA 3. *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. 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). From Lemma 1, servers in $S1$ (and hence, in $S12$) reply $ts \geq k$ to rd because write(k) precedes rd). Thus, the highest timestamp received by rd, $z \geq k$. There are the following two cases to consider:

1. $z > k$
   Since rd returns either z or $z - 1$, it follows that rd does not return a timestamp lower than 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, $r_j \in m.seen$. Thus, $\{w, r_j\} \subseteq \cap_{m \in MS} m.seen$. As $|S12| \geq S - 2t$, in rd, the predicate in line 19 holds with $a = 2$. Consequently, rd returns $z = k$.

LEMMA 4. *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 z, 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 2, the read returns a value greater than or equal to x. Using Lemma 2 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, receive $S - t$ 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:
$\langle 1 \rangle 1$. the predicate in line 17 does not hold in $rd1$.
It follows that $x = TS1 - 1$. Thus some servers have 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 3, $rd2$ returns $y \geq x$.
$\langle 1 \rangle 2$. the predicate in line 17 holds in $rd1$.
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:
$\langle 2 \rangle 1$. $y = TS2$
$y = TS2$. Since, $|S12| > t$ and $|S2| = S - t$, there is a server $p_i \in S2 \cap S12$. Since $rd1$ precedes $rd2$, p_i first replies $ts = x$ to $rd1$ then replies to $rd2$. From Lemma 1, it follows that p_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.
$\langle 2 \rangle 2$. $y = TS2 - 1$
There are the following two subcases to consider:
$\langle 3 \rangle 1$. $y + 1 \neq x$
As in case $\langle 2 \rangle 1$, we can show that there is a server $p_i \in S2 \cap S12$, and p_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, $y \geq x$.
$\langle 3 \rangle 2$. $y + 1 = x$
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:

$\langle 4 \rangle 1$. $a \leq R$

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, $MS1 \subseteq MS$.[4] 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:

$\langle 5 \rangle 1$. $r_k \notin \cap_{m \in MS1} m.seen$

Let $MS2$ be the set of messages received by $rd2$ from processes in $S2 \cap S12$. For any server $p_i \in S2 \cap S12$, let $m1_i$ and $m2_i$ be the messages sent by p_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.

$\langle 5 \rangle 2$. $r_k \in \cap_{m \in MS1} m.seen$

Thus each server p_i in $S2 \cap S12$ has sent at least one READACK message with $ts = x$ to r_k, before p_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 p_i in $S2 \cap S12$ whose READACK message is received by $rd2a$, say message m. Now consider the last READACK message sent by p_i to r_k before $rd2$ is invoked, say message m'. Since we know that p_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 1 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 p_i checks $counter[k]$ before replying to r_k. Thus, once m is sent by p_i, $counter[k]$ at p_i is set such that p_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 p_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 $rd2a$ is greater than or equal to x. It follows that $rd2$ sends READ messages with $ts \geq x$. From Lemma 2, $rd2$ returns a timestamp greater than or equal to x. As $x = y+1$, $rd2$ does

[4] See case $\langle 1 \rangle 2$ for the definition of MS.

not return y, a contradiction.

$\langle 4 \rangle 2$. $a = R + 1$

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 $5\langle 2 \rangle$), we can show that $rd2$ returns a value greater than or equal to x, a contradiction.

5. LOWER BOUND

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

PROPOSITION 5. *Let $t \geq 1$ and $R \geq 2$. If $R \geq \frac{S}{t} - 2$, then there is no fast atomic register implementation.*

Preliminaries. Recall first that w denotes the writer, r_i for $1 \leq i \leq R$ denote the readers, and s_i for $1 \leq i \leq S$ denote the servers. 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 $(1 \leq i \leq R+2)$, each of size less than or equal to t.[5]

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 servers to the invoking processes, are delivered in those partial runs. 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 invocation inv 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 *inv* in that partial run, (2) all other servers receive the READ or the WRITE message from *inv* and reply to that message, and (3) *all these reply messages are in transit.* We say that a *complete invocation inv skips* a block B_i in a partial run, if (1) no server in B_i receives any READ or WRITE message from *inv* in that partial run, (2) all servers that are not in B_i receive the READ or WRITE message from *inv* and reply to that message, and (3) the invoking process *receives all these reply messages and returns from the invocation.*

[5]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 $B_{R+2} = \{s_j \mid (\lfloor \frac{S}{R+2} \rfloor (R+1)) \leq j \leq S\}$. 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 show the impossibility. The impossibility still holds if we add more readers to this system (i.e., $R > S - 2$).

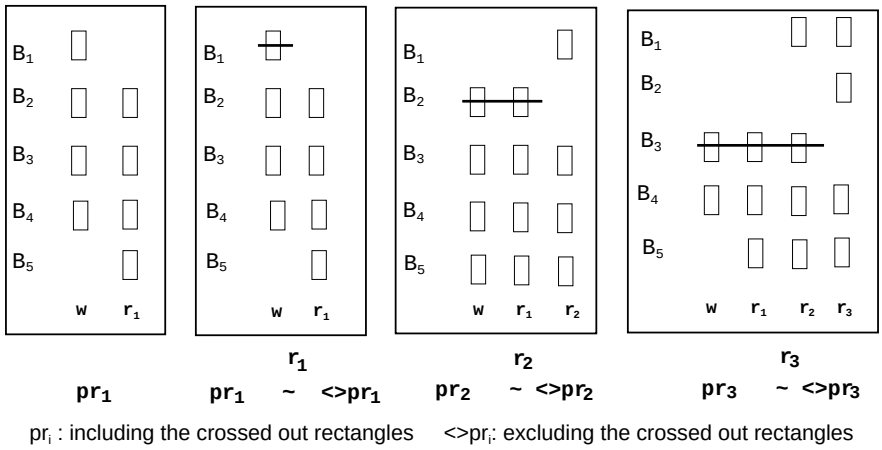


Fig. 3. Partial runs: pr_i and $\triangle pr_i$

To show 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 invocation 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 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) invocation that skips $\{B_{R+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. We make the following simple observations: (1) for $1 \leq i \leq R$, 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 returns from the write invocation, and hence, (3) wr and wr_1 differ only at w.

Block diagrams. We illustrate a particular instance of the proof in 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 an invocation inv through a set of rectangles, (generally) arranged in a single column. In the column corresponding to some invocation inv, we draw a rectangle in the i^{th} row, if all servers in block B_i have received the READ or WRITE message from inv and have sent reply messages, i.e., we draw a rectangle in the i^{th} row if inv does not skip B_i. (We present a slightly more detailed diagram of the partial writes in Figure 1 in the optional appendix.)

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 $\triangle pr_1$, that extends wr_2 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 these processes in

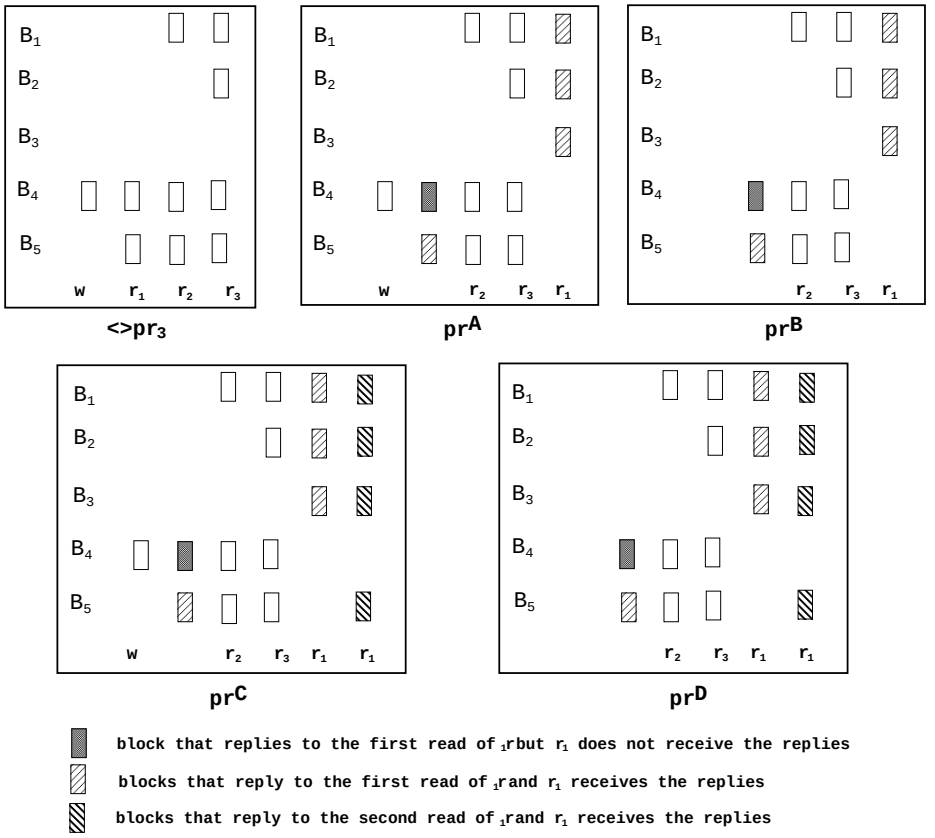


Fig. 4. Partial runs: pr^A, pr^B, pr^C and pr^D

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:[6] 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 $\triangle pr_R$.

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

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 $\{B_j | h \leq j \leq R\}$. 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. Observe that, in $\triangle pr_R$, only the servers in B_{R+1} receive the WRITE message from the write(1) invocation. 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 these replies from servers in B_1 to B_R, and then r_1 returns from the read invocation. (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 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 other readers. By atomicity, r_1's read returns (the initial value of the register) $\perp$ in pr^B because there is no write($*$) invocation 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 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($*$) invocation 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.

6. ARBITRARY FAILURE MODEL

An arbitrary failure can either correspond to a crash or a malicious behavior. A process is malicious if it deviates from the algorithm assigned to it in a way that is different from simply stopping all activities (crashing). We distinguish two resilience thresholds: b and t [Lamport 2003]. Just as in the crash-stop model, a maximum number of t processes can crash. However, out of these t processes, up to b processes can be malicious. We therefore always have $b \leq t$.

In the literature, the special case where $b = t$ is usually considered. There are several advantages of distinguishing these two thresholds. First, we highlight the influence on the lower bounds of (1) the malicious behavior of the processes and (2)

1: at the writer w
2: **procedure** initialization:
3: $ts \leftarrow 1$, $rCounter \leftarrow 0$
4: **procedure** write(v)
5: send (WRITE, $ts_{\sigma w}$, $rCounter$) to all servers
6: **wait until** receivevalid(WRITEACK, $ts_{\sigma w}$, $*$, $rCounter$) from $S - t$ servers
7: $ts \leftarrow ts + 1$
8: return(OK)

9: at each reader r_i
10: **procedure** initialization:
11: $ts \leftarrow 0$; $rCounter \leftarrow 0$; $maxTS_{sgn} \leftarrow 0$
12: **procedure** read()
13: $rCounter \leftarrow rCounter + 1$; $ts \leftarrow maxTS_{sgn}$
14: send(READ, ts, $rCounter$) to all servers
15: **wait until** receivevalid(READACK, $ts'_{\sigma w}$, $seen'$, $rCounter$), $ts' \geq ts$, $r_i \in seen'$ from $S - t$ servers
16: $rcvMsg \leftarrow \{m | r_i$ received (READACK, $*$, $*$, $rCounter)\}$
17: $maxTS_{sgn} \leftarrow ts_{1\sigma w} \mid ts_1 = \mathbf{Maximum}\{ts' |$ (READACK, $\langle ts' \rangle_{\sigma w}$, $*$, $rCounter) \in rcvMsg\}$
18: $maxTSmsg \leftarrow \{m | m.ts = maxTS_{sgn}$ **and** $m \in rcvMsg\}$
19: **if** there is $a \in [1, R+1]$ and there is $MS \subseteq maxTSmsg$ s.t., $(|MS| \geq S - at - (a-1)b)$ **and** $(| \cap_{m \in MS} m.seen| \geq a)$ **then**
20: return($maxTS$)
21: **else**
22: return($maxTS - 1$)

23: at each server p_i
24: **procedure** initialization:
25: $ts_{sgn}, ts \leftarrow 0$; $seen \leftarrow \emptyset$; $counter[0...R] \leftarrow [0...0]$
26: **upon** receivevalid($msgType$, $ts'_{\sigma w}$, $rCounter'$) from $q \in \{w, r_1, ..., r_R\}$ **and** $rCounter' \geq counter[pid(q)]$ **do**
27: **if** $ts' > ts$ **then**
28: $ts \leftarrow ts'$; $ts_{sgn} \leftarrow ts'_{\sigma w}$; $seen \leftarrow \{q\}$
29: **else**
30: $seen \leftarrow seen \cup \{q\}$
31: $counter[pid(q)] \leftarrow rCounter'$
32: **if** $msgType =$ READ **then**
33: send(READACK, ts_{sgn}, $seen$, $rCounter'$) to q
34: **else**
35: send(WRITEACK, ts_{sgn}, $seen$, $rCounter'$) to q

Fig. 5. Fast SWMR atomic register implementation with $S > (R+2)t + (R+1)b$

the processes' non-responsiveness. In the boundary case, where $b = 0$, the failure model becomes non-arbitrary, i.e., the traditional crash failure model. Hence, the bounds derived in this paper for the general case with parameters b and t bridge the gap between arbitrary and crash failure models, by establishing a result that is applicable to both.

6.1 A Fast Implementation

We describe in this section a fast implementation in the arbitrary failure model assuming $S > (R+2)t + (R+1)b$ which is equivalent to $R < \frac{S+b}{t+b} - 2$ (Figure 5).

The algorithm is similar to the one presented in Section 4 except for a few key differences. First of all, the writer digitally signs each value [Rivest et al. 1978]. Digital signatures allow us to make the following assumptions:

PROPERTY 1. **Authentication:** *readers can check that a value returned by a server was in fact written by the writer.*

PROPERTY 2. **Unforgeability:** *it is impossible to forge the digital signature of the writer.*

Apart from the addition of digital signatures, the write mechanism is unchanged and the writer waits for the response of $S - t$ servers.

Our read procedure begins with servers issuing a READ message containing the highest signed timestamp encountered in the previous read invocation (lines 13-14). In a way, the reader writes back this timestamp, signed by the writer ($ts_{\sigma w}$), to all servers. During the first read invocation, the reader issues a read message with the default timestamp 0, which is also the initial timestamp at servers and writer. We assume that this initial value is not digitally signed by the writer. Then, the reader collects responses from $S-t$ servers containing the latest timestamps signed by the writer encountered by the servers (including the one being written back by the reader). The reader then selects the highest timestamp, denoted by $maxTS_{sgn}$ in Figure 2 (that will be written back by the reader in its next read invocation).

The mechanism of the read procedure is also very similar, except for the predicate which checks if the latest value has been seen by a sufficient number of processes. 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, out of which at least $S-t-b$ are non-malicious, let this set of servers be S_1. Subsequently, a reader reads from a set S_2 (of $S - t$ servers) that overlaps at $S - 2t - b$ (non-malicious) 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, with a failure pattern different from pr_1, the write is incomplete and the writer writes ts only to $S - 2t - b$ servers (possibly malicious) 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 $S1 \cap S2$, and accounting for the possibility that another b servers are malicious, it is easy to see that the new read has to return ts, even if it sees ts at $S - 3t - 2b$ servers that have already replied to both the write and the first read. Thus the predicate for the read procedure is as follows: if there is $a \geq 1$ such that the reader receives $maxTS_{sgn}$ in at least $S - at - (a - 1)b$ messages, and there are at least a processes that are in the list *seen* of each of these $S - at - (a - 1)b$ messages, then the predicate is true.

We now prove the correctness of the fast implementation depicted in Figure 5. We do not assume that the lines in Figure 5 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. We assume that, if a process receives an incomplete message, the process can detect that the message is incomplete, and ignores such a message.

It is obvious that read and write procedures complete in one round-trip. To show atomicity, we recall that the write procedure directly writes the timestamp. Thus the conditions in Section 3.1 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 to 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 condition is trivial and it is not difficult to see that the third property holds, having in mind the unforgeability property of the writer's digital signature. Below, we show the other two. In the proofs we refer to the global clock; however processes do not access this global clock.

LEMMA 6. *If a non-malicious server s sets ts to x at time T, then s never sets ts to a value that is lower than x after time T.*

PROOF: obvious from line 33.

LEMMA 7. *If a read() sends* READ *messages with $ts = x$, then the read does not return a value smaller than x.*

PROOF: recall that a read should return a value only if at most t servers are faulty (liveness). Suppose read rd by r_i sends a READ message with $ts = x$. From line 33, every READACK message received by rd from a non-malicious server is with $ts \geq x$. Reader awaits for $S - t$ READACK messages before returning a value. Moreover, reader discards all READACK messages that have a timestamp less than x, as those READACK messages are clearly from malicious servers. Eventually, if there are at most t server failures, rd receives READACK messages from $S - t$ non-malicious servers. Let z be the maximum of those timestamps (i.e., $maxTS_{sgn}$ computed in line 20). Clearly, $z \geq x$. 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 $ts = x$ (as all READACK that are received, and not discarded, are from non-malicious servers) and has $r_i \in seen$. Since rd receives $S - t$ READACK messages from non-malicious servers, the predicate in line 22 of rd holds with $a = 1$. Hence, rd returns x.

LEMMA 8. *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 7, and let $S2$ be the set of $S - t$ servers from which rd received READACK messages in line 18. Let $S12 = S1 \cap S2$ and $S12_{nm}$ a subset of $S12$ that contains only non-malicious servers. Obviously, $|S12_{nm}| \geq S - 2t - b$. Let z be the maximum timestamp received by rd from servers in $S2$. Observe that rd returns either z or $z - 1$.

When a non-malicious server in $S1$ replies to a WRITE message from wr, its ts is k. 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). From Lemma 6, non-malicious servers in $S1$ (and hence, in $S12_{nm}$) reply $ts \geq k$ to rd because write(k) precedes rd). Thus, the highest timestamp received by rd, $z \geq k$. There are the following two cases to consider:

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

—$z = k$
We know that every (non-malicious) server in $S12_{nm}$ replies to rd with $ts \geq k$, and $z = k$ is the maximum timestamp received by rd from servers in $S2 \supseteq S12 \supseteq S12_{nm}$. Thus every server in $S12_{nm}$ replies $ts = k$ to rd. Let MS be the set of READACK messages sent by servers in $S12_{nm}$ to rd. Since every server in $S12_{nm}$ replies $ts = k$ to wr before sending $ts = k$ to rd, for every message m in MS, $w \in m.seen$. Furthermore, from line 36, $r_j \in m.seen$. Thus, $\{w, r_j\} \subseteq \cap_{m \in MS} m.seen$. As $|S12_{nm}| \geq S - 2t - b$, in rd, the predicate in line 22 holds with $a = 2$. Consequently, rd returns $z = k$.

LEMMA 9. *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 z, 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 7, the read returns a value greater than or equal to x. Using Lemma 7 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, receive $S - t$ READACK messages in line 18. Let $TS1$ be the highest timestamp received by $rd1$ from processes in $S1$ (i.e., the $maxTS_{sgn}$ evaluated in line 20of $rd1$). Similarly, let $TS2$ be the highest timestamp received by $rd2$ from the processes in $S2$. There are the following two cases to consider:
⟨1⟩1. the predicate in line 20 does not hold in $rd1$.
It follows that $x = TS1 - 1$. Thus some servers have 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 8, $rd2$ returns $y \geq x$.
⟨1⟩2. the predicate in line 20 holds in $rd1$.
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 $S - at - (a-1)b$ 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 $S > (R+2)t + (R+1)b$, $|S12| = |MS| = S - at - (a-1)b > t + b$. There are the following two cases to

consider:

⟨2⟩1. $y = TS2$

$y = TS2$. Since, $|S12| > t + b$ and $|S2| = S - t$, there is a non-malicious server $p_i \in S2 \cap S12$. Since $rd1$ precedes $rd2$, p_i first replies $ts = x$ to $rd1$ then replies to $rd2$. From Lemma 6, it follows that p_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.

⟨2⟩2. $y = TS2 - 1$

There are the following two subcases to consider:

⟨3⟩1. $y + 1 \neq x$

As in case ⟨2⟩1, we can show that there is a non-malicious server $p_i \in S2 \cap S12$, and p_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, $y \geq x$.

⟨3⟩2. $y + 1 = x$

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

⟨4⟩1. $a \leq R$

Then $|S2 \cap S12 \cap Snm| \geq S - (a + 1)t - ab > t + b$. Let $MS1$ be the set of READACK messages from processes in $S2 \cap S12 \cap Snm$ (non-malicious processes from $S1 \cap S12$) to $rd1$. From the definition of $MS1$ and MS, $MS1 \subseteq MS$.[7] 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:

⟨5⟩1. $r_k \notin \cap_{m \in MS1} m.seen$

Let $MS2$ be the set of messages received by $rd2$ from processes in $S2 \cap S12 \cap Snm$. For any (non-malicious) server $p_i \in S2 \cap S12 \cap Snm$, let $m1_i$ and $m2_i$ be the messages sent by p_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 $|S2 \cap S12 \cap Snm| \geq S - (a+1)t - ab$, the number of message in $MS2$ is at least $S - (a + 1)t - ab$. As $a + 1 \leq R + 1$, the predicate in line 22 in $rd2$ holds with $a + 1$. Thus, the timestamp returned by $rd2$ is $x = y + 1$, a contradiction.

⟨5⟩2. $r_k \in \cap_{m \in MS1} m.seen$

Thus each server p_i in $S2 \cap S12 \cap Snm$ has sent at least one READACK message with $ts = x$ to r_k, before p_i sent the $MS1$ message to r_j. Since the messages in $MS1$ are sent before the completion of $rd1$ (and

[7] See case ⟨1⟩2 for the definition of MS.

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 $|S2 \cap S12 \cap Snm| \geq S - (a+1)t - ab > t + b$, there is at least one process p_i in $S2 \cap S12 \cap Snm$ whose READACK message is received by $rd2a$, say message m. Now consider the last READACK message sent by p_i to r_k before $rd2$ is invoked, say message m'. Since we know that p_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 6 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 p_i checks $counter[k]$ before replying to r_k. Thus, once m is sent by p_i, $counter[k]$ at p_i is set such that p_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 p_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 $rd2a$ is greater than or equal to x. It follows that $rd2$ sends READ messages with $ts \geq x$. From Lemma 7, $rd2$ returns a timestamp greater than or equal to x. As $x = y + 1$, $rd2$ does not return y, a contradiction.

$\langle 4 \rangle 2$. $a = R + 1$

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 $|S12| \geq S - at - (a-1)b > t + b$. (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 $5\langle 2 \rangle$), we can show that $rd2$ returns a value greater than or equal to x, a contradiction.

6.2 Optimality

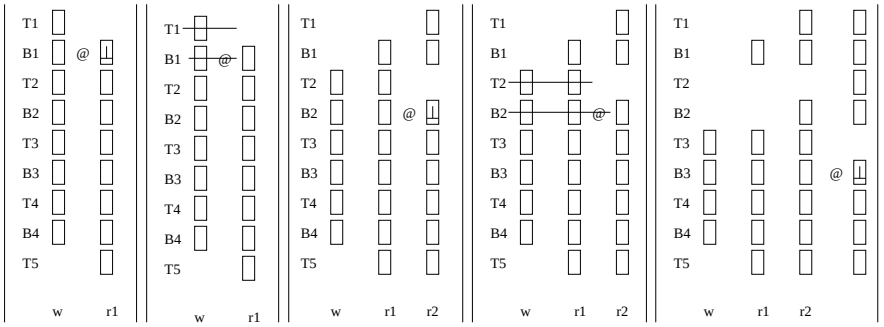
The following proposition states that the resilience required by our fast implementation is indeed necessary.

PROPOSITION 10. *Let $t \geq 1$, $b \geq 0$ and $R \geq 2$. If $(R+2)t + (R+1)b \geq S$, then there is no fast atomic register implementation.*

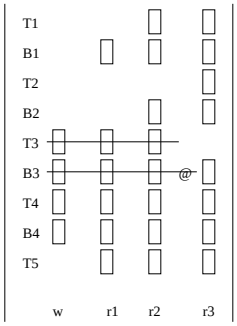
This proof is similar to the one in Section 5: we suppose by contradiction that $(R + 2)t + (R + 1)b \geq S$ and that there is a fast implementation I of an atomic register (even with public key cryptography). We construct a partial run of the fast implementation I that violates atomicity: a partial run in which some read returns 1 and a subsequent read returns an older value, namely, the initial value of the register, $\perp$. This run is different from the one in the previous proof.

Partial writes. Note that we assume that writer writes digitally signed information to servers, and that arbitrarily faulty servers cannot forge the writers signature. Consider a partial run wr in which w completes write(1) on the register. The invo-

cation skips T_{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 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) invocation that skips $\{T_{R+2}\} \cup \{T_j | 1 \leq j \leq i-1\} \cup \{B_j | 1 \leq j \leq i-1\}$. We make the following simple observations: (1) for $1 \leq i \leq R$, wr_i and wr_{i+1} differ only at servers in $T_i \cup B_i$, (2) wr is an extension of wr_1, such that, in wr, w receives the replies (that are in transit in wr_1) and returns from the write invocation, and hence, (3) wr and wr_1 differ only at w.



(a) pr1 (b) $pr1 - \triangle pr1$ (c) $\triangle pr1 - pr2$ (d) $pr2 - \triangle pr2$ (e) $\triangle pr2 - pr3$



(f) $pr3 - \triangle pr3$

Fig. 6. Partial runs pr_i and $\triangle pr_i$

Block diagrams. We illustrate a particular instance of the proof in Figure 6 and Figure 4 (unchanged from Section 5), where $R = 3$ and the set of servers are partitioned into nine blocks, T_1 to T_5 and B_1 to B_4. We depict an invocation inv through a set of rectangles, (generally) arranged in a single column. In the column corresponding to some invocation inv, we draw a rectangle in the i^{th} row, if all servers in block B_i have received the READ or WRITE message from inv and have sent reply messages, i.e., we draw a rectangle in the i^{th} row if inv does not skip B_i. In partial run pr_i, we denote the failure of B_i by @ (B_i "looses its memory").

Appending reads. Partial run pr_1 extends wr by having block B_1 failing arbitrarily upon completion of write(1) and appending a complete read by r_1 that skips block T_1. B_1 fails in such a way that it behaves as if it never any message (i.e., a message from invocation write(1)). We say that B_1 fails and loses its memory. Observe that r_1 cannot distinguish pr_1 from some partial run $\triangle pr_1$, that extends wr_2 by appending a complete read by r_1 that skips T_1. To see why, notice that wr and wr_2 differ at w and at blocks T_1 and B_1, and r_1 does not receive any message from process w and block T_1 in both runs and it received the same message from block B_1 in both runs. By liveness property, r_1's read in $\triangle pr_1$ must return some value x (as it cannot wait for the completion of the writer's invocation, nor a message from w). As r_1 cannot distinguish $\triangle pr_1$ from pr_1, it returns the same value x in pr_1 as well, and by atomicity, in pr_1 x must equal 1. Therefore, in $\triangle pr_1$ r_1 also returns 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: (1) block B_i failing arbitrarily in such a way that it behaves as if it never received any message (loses memory) and (2) appending a complete read by r_i that skips T_i. Partial run $\triangle pr_i$ is constructed by deleting from pr_i, all steps of the servers in block T_i and all steps of servers in block B_i up to the instant in which B_i lost its memory (including that particular step). Since the last read in pr_i by reader r_i skips block T_i, r_i cannot distinguish pr_i from $\triangle pr_i$, as in both runs r_i receives the same messages from B_i. More precisely, partial run $\triangle pr_i$ extends wr_{i+1} by appending the following i reads one after the other:[8] for $1 \leq h \leq i-1$, r_h does a read that skips $\{T_j | h \leq j \leq i\} \cup \{B_j | h+1 \leq j \leq i\}$ and r_i does a (complete) read that skips T_i. Figure 6 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. By liveness requirements in $\triangle pr_2$ r_2 must return some value, say x_2. However, as r_2 cannot distinguish pr_2 from $\triangle pr_2$, so it must return a value x_2 in pr_2, as well. Since pr_2 extends $\triangle pr_1$, by atomicity, r_2's read in pr_2 must return $x_2 = 1$. Therefore, 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 $2 \leq i \leq R$), in which it must return a value, it follows from a trivial induction that r_i's read in $\triangle pr_i$ returns 1. In particular, r_R reads 1 in $\triangle pr_R$. Moreover, note that in $\triangle pr_R$ no server is faulty.

Partial run pr^A. Consider the partial run $\triangle pr_R$: wr_{R+1} extended by appending R reads by each reader r_h $(1 \leq h \leq R-1)$ such that r_h's read skips $\{T_j | h \leq j \leq R-1\} \cup \{B_j | h+1 \leq j \leq R-1\}$ and a read by reader r_R skips T_R only. The read by r_1 is incomplete in $\triangle pr_R$: only servers in B_1, T_{R+1}, B_{R+1} and T_{R+2} send replies to r_1, and those reply messages are in transit. Observe that, in $\triangle pr_R$, only the servers in T_{R+1} and B_{R+1} receive the WRITE message from the write(1) invocation. Consider the following partial run pr^A which differs from the $\triangle pr_R$ in the following: (0) Upon reception of message from write(1) invocation, B_{R+1} fails arbitrarily in such a way that, from that point on, it sends replies to all processes but r_1 as if it was not faulty, and to r_1 as if it never received a write(1) message.

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

Moreover, after completion of read by r_R, (1) r_1 receives the replies of its READ messages from T_{R+2} and B_1 (that were in transit in $\triangle pr_R$) and B_{R+1} (the faulty ones), (2) the servers in T_1 to T_R and B_2 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 these replies from servers in T_1 to T_R and B_2 to B_R, and then r_1 returns from the read invocation. (Notice that, r_1 received replies from all blocks but T_{R+1}, and so, must return from the read. However, r_1 does not receive the replies from servers in T_{R+1}.

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 block B_{R+1} is not faulty. Hence, servers in T_{R+1} do not receive any WRITE message (Figure 4). Clearly, only servers in T_{R+1}, B_{R+1}, the writer, and the readers 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 T_{R+1}, the writer, or other readers and it receives the same message from the servers in B_{R+1} in both runs. By atomicity, r_1's read returns (the initial value of the register) $\perp$ in pr^B because there is no write($*$) invocation 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 T_{R+1}, and (2) pr^D is constructed by extending pr^B with another complete read by r_1, which skips T_{R+1} (Figure 4). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips T_{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($*$) invocation 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.

7. MULTIPLE WRITERS

The atomicity definition presented in Section 3 extends to multi-writer multi-reader (MWMR) registers as well. In the impossibility proof below, we use two simple properties of MWMR atomic register which can be easily deduced from atomicity. In any partial run (property **P1**) if a write wr that writes v, precedes some read rd, and all other writes precede wr, then if rd returns, it returns v, and (property **P2**) if there are two reads such that all writes precede both reads, then the reads do not return different values.

The proposition below states that there cannot exist a fast multi-writer atomic register implementation. The proof is written for the crash-stop model. But by extension the impossibility directly applies to the malicious failure model.

PROPOSITION 11. *Let $t \geq 1$, $W \geq 2$, $R \geq 2$. Any atomic register implementation has a run in which some complete read or write is not fast.*

PROOF: it is sufficient to show the impossibility in a system where $W = R = 2$, and $t = 1$. 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 servers to the invoking processes, are delivered in those partial runs. 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 invocation *inv skips* a server s_i in a partial run if every server distinct from s_i receives the READ or the WRITE message from *inv* and replies to that message, *inv* receives those replies and returns, and all other messages are in transit. In other words, only s_i does not receive READ or WRITE message from *inv*. Since $t = 1$, any complete invocation may skip at most one server. If a complete invocation does not skip any servers, we say that the invocation is *skip-free*.

Consider a partial run run_1 constructed with the following three non-overlapping invocations: (1) a skip-free write(2) by w_2, that precedes (2) a skip-free write(1) by w_1, that in turn precedes (3) 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: (1) a skip-free write(1) by w_1, that precedes (2) a skip-free write(2) by w_2, that in turn precedes (3) 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., the replies of s_i are sent in the opposite order in r^{i+1} from that in run^1). Since servers do not receive any message 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, and hence, r_1 can not distinguish between 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 $r1$ returns 1 in run^1, 2 in r^{S+1}, and 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 $r1$ 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. Similarly we 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,

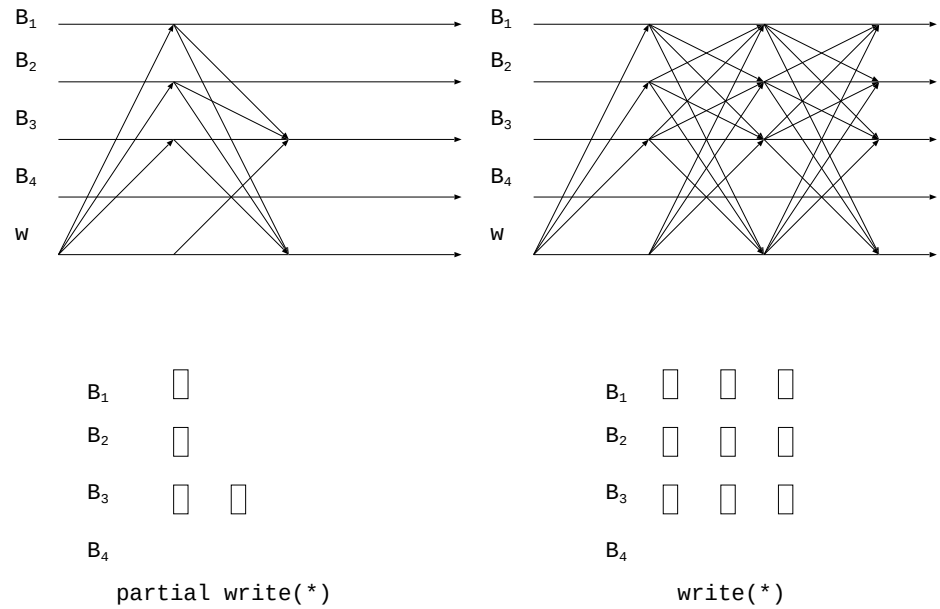


Fig. 7. Partial writes ($K = 3, R = 4$)

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, in returns 2 in run'' as well. Clearly, run^{i1+1} violates property P2.

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

8. WHEN "ATOMIC READS MUST WRITE"

Our results revisit, in a message passing context, the folklore theorem that "atomic reads must write", borrowed from the shared-memory context [Lamport 1986; Attiya and Welch 1998]. In particular, a result from [Attiya and Welch 1998] 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*[9] ones [Lamport 1986], 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, [Fan and Lynch 2003] has shown that, in a message-passing system,

[9]A regular register is like an atomic register except when there is concurrency: a a reader might not return the last value written.

every atomic read must modify the state of at least t servers, which might be interpreted as a need for a second communication round-trip. 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. In fact, processes (servers) are smarter than basic (regular or single-reader) registers and might intuitively do a lot in one communication round-trip.

Our results also draw a sharp line between the time-complexity of regular [Lamport 1978] and atomic register implementations. For instance, in the crash-only model, 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). In this model, we show that a fast implementation of a SWMR atomic register exists if and only if $t < \frac{S}{R+2}$. However, since fast atomic registers have exactly the same time-complexity as regular registers they are clearly the most interesting option where only a few readers are necessary. In an application where a high number of readers (or writers) are required a trade-off needs to be made. A regular register will provide speed at the expense of consistency and an atomic register will provide better consistency guarantees at the expense of speed.

9. SUMMARY

This paper establishes the exact conditions required for a fast implementation of an atomic read-write data structure, also called a register.

In the case of multiple writers, we proved that a fast implementation is impossible even if only one server can fail, and it can only do by crashing.

In the case of a single-writer where t out of S servers can fail by crashing, the number of readers must be smaller than $S/t - 2$. In the general arbitrary failure model, this number must be smaller than $(S + b)/(t + b) - 2$ where up to b out of t servers can be malicious.

REFERENCES

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

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

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

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

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

Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–565.

Lamport, L. 1986. On interprocess communication. *Distributed computing 1,* 1 (May), 77–101.

Lamport, L. 2003. Lower bounds for asynchronous consensus. *Future Directions in Distributed Computing*, 22–23.

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

Lynch, N. A. 1996. *Distributed Algorithms.* Morgan-Kaufmann.

Rivest, R. L., Shamir, A., and Adleman, L. M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21,* 2, 120–126.