FAST ACCESS TO DISTRIBUTED ATOMIC MEMORY *

PARTHA DUTTA[†], RACHID GUERRAOUI[‡], RON R. LEVY[§], AND MARKO VUKOLIĆ[¶]

Abstract. 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 a number of servers implementing the data structure. We determine the exact conditions under which every read and write involves *one round* of communication with the servers. These conditions relate the number of readers to the tolerated number of faulty servers and the nature of these failures.

Key words. Atomic registers, Byzantine failures, Distributed algorithms, Fault-tolerance, Time-complexity, Shared-memory emulations.

1. Introduction.

1.1. Background. Assigning a value to a variable or 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* or simply a *register*, is fundamental in distributed computing and is at the heart of a large number of distributed algorithms [16, 5].

We study distributed implementations of this abstraction in an asynchronous message-passing system with no actual physical shared memory: instead, 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 client processes. We consider *robust* [4] implementations where any read or write invocation by some client process eventually returns, independently of (a) the operational status of other clients: all of them might have stopped their computation (this aspect of robustness is also called *wait-freedom* [16]); and (b) the failure of some of the servers. Such implementations have recently attracted a lot of interest as they underly reliable distributed storage systems [25, 2, 17], which constitute appealing alternatives to traditional centralized storage infrastructures.

Ensuring both atomicity and robustness is not trivial. Informally, atomicity requires that, even though each read or write operation may overlap and take an arbitrary period of time to complete, they appear to execute at some indivisible instant during their respective period of execution [18]. This requires ordering operations in a way that respects their real-time order as well as the expected sequential specification of 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 fast implementations, consider the classical implementation

*This work is based on an earlier work: "How Fast can a Distributed Atomic Read be?" in The Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC '04). ©ACM, 2004. http://doi.acm.org/10.1145/1011767.1011802

[†]IBM Research - India, India (parthdut@in.ibm.com).

[‡]School of Computer and Communication Sciences, EPFL, Switzerland (rachid.guerraoui@epfl.ch).

[§]Quorius Ltd., Switzerland (ron.levy@gmail.com).

[¶]IBM Research - Zurich, Switzerland (mvu@zurich.ibm.com).

from [4] in a single-writer multi-reader case, also called a SWMR register implementation [18]. In [4], readers and servers are the same set, the writer is one of the servers, a minority of processes may fail by crashing, i.e., halting all their activities without warning, whereas the rest of the processes execute the algorithm assigned to them.

This implementation ensures atomicity 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. Given the single-writer setting, and since only the writer introduces new timestamps in the system, the writer always knows the latest timestamp. Unlike the writer, 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: a subsequent read might thus miss this value. 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 scheme 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 at least 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 at least 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 round-trip of communication between a client and servers?

This would clearly be optimal in terms of time-complexity. Besides theoretical interest, such implementation might for instance be of practical relevance in the context of distributed storage systems where fast access to shared information might be of primary importance [1, 7, 29].

Clearly, the difficulty is related to the multiplicity of readers. With a single reader, it is easy to modify the algorithm of [4] such that the read takes only one round-trip [18]: 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 and ensures atomicity. To illustrate this case,

suppose the writer writes v with timestamp ts, 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 two readers. 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 ts, 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 any minority of servers can be faulty. But what if we further restrict the number of tolerated server failures?

1.2. Contributions. We show in this paper that, interestingly, the existence of a fast SWMR implementation depends on the maximum number R of readers. Notice here the focus on SWMR implementations; in this paper we, therefore, assume $R \geq 2$. We consider a general model of computation where t among the set S of server processes on which the data structure is implemented can fail; in this paper, we assume $t \geq 1$. A server can fail by crashing, or even by deviating arbitrarily from its algorithm and be malicious (also called Byzantine [26]). We denote by b the number of arbitrary server failures, where $0 \leq b \leq t$. In this paper, we consider the authenticated arbitrary failure model in which processes can rely on unforgeable digital signatures [28].

The main contribution of this paper is a theorem stating the following:

THEOREM 1.1. *There is a fast implementation of a SWMR register if and only if the number of readers R is less than $\frac{S+b}{t+b} - 2$.*

The paper proves this theorem by giving a fast implementation and then proving it optimal (in terms of number of readers). Both the algorithm and the lower bound proof are, we believe, interesting in their own rights. The algorithm uses a new *trace-based* memory access technique whereas the lower bound uses a *sieve-based* run construction scheme.

- **Algorithm.** Our fast implementation relies on the idea of *traces* left by readers in the servers they access, even if they expedite their operation in one round-trip. These traces are then used to determine which value to return while preserving atomicity. It is important to notice at this point that this idea has not been used in the register transformations literature e.g., [4] because, in a read/write shared memory, a process cannot read a value and at the same time leave a trace. We exploit the idea to obtain a crash-stop fast implementation (i.e., assuming $b = 0$ and $R < \frac{S}{t} - 2$) and then a Byzantine-resilient one (i.e., assuming $b \neq 0$).

  To get an intuition of the idea in the crash-stop case, consider the classical algorithm of [4], sketched above, 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 cannot be simply deduced from the number of servers that has seen the value. To determine whether a value is safe to return, we make every server maintain, besides the latest value, the set of readers to which the server has sent that value. This is the actual *trace* left by the readers.

- **Lower bound.** Consider S server processes, t among which can be faulty, and none is Byzantine. We prove by contradiction that there is no fast implementation with $R \geq \frac{S}{t} - 2$ (recall here that we assume $R \geq 2$ and $t \geq 1$). We illustrate the proof in Figure 1.1 for $S = 4$, $t = 1$ and $R = 2$.

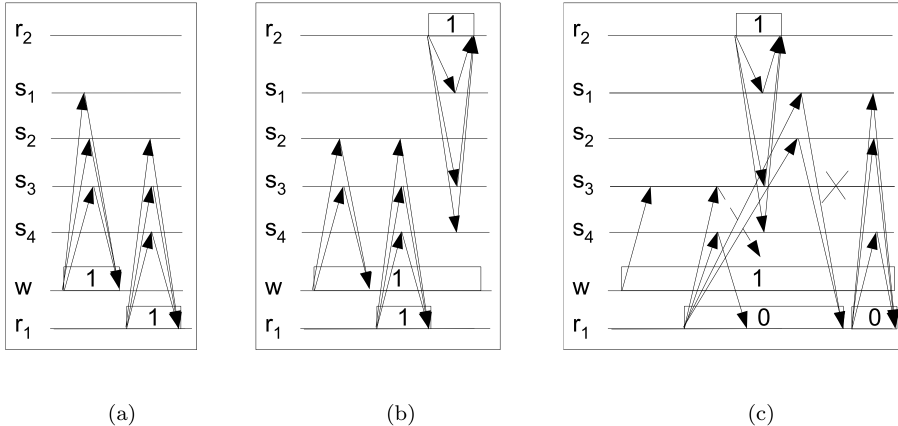


FIG. 1.1. *Sketch of the lower bound proof for $S = 4$, $t = 1$ and $R = 2$*

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, to which we append a read that misses t other servers (see Fig. 1.1(a)). 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, each acting like a *sieve*, 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 (Fig. 1.1(b)). 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. In the special case shown in Figure 1.1(c)) we obtain atomicity violation by reusing the first reader r_1: a) the first read invocation by r_1, concurrent with that of reader r_2, cannot return 1 because it cannot read value 1 from any server (the response from s_3 is never received since s_3 crashes) and b) for similar reasons, the following read by r_1 cannot return 1, however a preceding read by r_2 already returned 1 — an atomicity violation.

The lower bound proof is then extended to the case where $b \leq t$ servers can fail in an arbitrary manner: we show that a fast implementation is possible *only if* the number of readers is less than $\frac{S+b}{t+b} - 2$.

Finally, to complete the picture, we prove that it is impossible to have a *one-round*

read algorithm with multiple writers [20] (MWMR atomic register) even if only one server can fail and it can only do so by crashing.

1.3. Roadmap. The paper is organized as follows. Section 2 gives the system model. Section 3 defines atomic (and fast) register 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. We discuss related work in Section 8. Section 9 summarizes the main results of the paper.

2. System Model.

2.1. Basics. The distributed system we consider consists of three *disjoint* sets of processes: a set *servers* (also denoted by Σ) of cardinality S containing processes $\{s_1, ..., s_S\}$, a set *writer* containing a single process $\{w\}$ (we discuss the multi-writer case in Section 7), and a set *readers* of cardinality R containing processes $\{r_1, ..., r_R\}$. We refer to the elements from $writer \cup readers$ as *clients*. We also denote a set $\Sigma \setminus \Sigma'$, where $\Sigma' \subseteq \Sigma$ by $\overline{\Sigma'}$. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel. Notice that the reliable channels assumption is needed to ensure *wait-freedom*, but not *atomicity* (we define these notions later). We also assume the existence of a global clock; however, processes cannot access the global clock.

A distributed algorithm A is a collection of automata, where A_p is the automata assigned to process p [22]. Computation proceeds in *steps* of A. 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. We extend our failure model to allow for arbitrary failures in Section 6.

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 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) (*reliable channels*) if there is a step that puts a message m in $mset$ such that p is the receiver of m and both p and the sender of m take an infinite number of steps, then there is a 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. Notice that the assumption of reliable

channels does not guarantee that correct processes always receive messages sent by faulty processes.

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 an operation 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).

In any run, we say that an operation $op1$ *precedes* operation $op2$ (or $op2$ *follows* $op1$) 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 *complete* in a (partial) run if the run contains a response step for that operation. We assume that all (partial) runs are *well-formed*, i.e., no process p invokes a new operation before all operations previously invoked by p have completed.

A *history* of a partial run is a sequence of invocation and response steps of operations in the same order as they appear in the partial run. An *incomplete* operation in a history H is an operation whose invocation step is in H, but the matching response step is not in H. We say that a history $H1$ *completes* history $H2$ if $H1$ can be obtained through the following modification of $H2$: for each incomplete operation op in $H2$, either invocation step of op is removed from $H2$, or any valid matching response for that invocation is appended to the end of $H2$.

3. Atomic Register. A sequential register is a data structure accessed by multiple processes in a non-concurrent manner (i.e., no two operations on a sequential register are concurrent). It provides two operations: write(v), which stores v in the register, and read(), which returns the last value stored. Only readers invoke reads on the register and only the writer invokes writes on the register. We further assume that there is a special write wr_0 operation that initializes the register by writing a special value $\perp$ (which is not a valid input value for other writes) such that wr_0 precedes all other operations.

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. An algorithm *implements* an atomic register if every run of the algorithm satisfies *termination* and *atomicity* properties, defined in the following.

3.1. Definition. Termination states that if a correct process invokes an operation, then eventually the operation completes (even if all other client processes have crashed).

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 A1-A3 below (Lemma 13.16 of [21]). 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: (A1) if $op1$ precedes $op2$ in H then it is not the case that $op2 \prec op1$, (A2) 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 (A3) the value returned by each read operation is the value written by the last preceding write operation according to $\prec$.

In our single writer setting assuming well-formed runs, atomicity properties A1-A3 can be simplified. Namely, in the single-writer case, the relation *precedes* totally orders write operations (and, likewise, the values stored by these). Before we give

the equivalent definition of atomicity in the single writer setting, we introduce some additional notation.

In a given run, we denote by wr_k the write that is preceded by exactly k writes (including the initial write wr_0). In other words, for $k \geq 1$, wr_k denotes the kth write by the writer in a given run (note that wr_0 is not invoked by the writer). Then, we say that an operation op returns *a timestamp* k, if: a) op is wr_k, or b) op is a read that returns the value stored by wr_k.

Consider a relation $\prec$ such that $op_1 \prec op_2$ if and only if the timestamp returned by op_1 is *smaller* than the timestamp returned by op_2. Then, it is straightforward to show that the $\prec$ is a partial ordering that satisfies atomicity properties A1-A3, if the following properties are satisfied:

(SWA1) If a read returns, it returns a non-negative timestamp.
(SWA2) If a read rd returns timestamp l and rd follows write wr_k, then $l \geq k$.
(SWA3) If a read rd returns timestamp k, then rd does not precede wr_k.
(SWA4) If reads rd_1 and rd_2 return timestamps l_1 and l_2, respectively, and if rd_2 follows rd_1, then $l_2 \geq l_1$.

Indeed, it is straightforward to see that Property A1 of $\prec$ is implied by Properties SWA2 and SWA4, whereas property A3 is implied by properties SWA1, SWA2 and SWA3. Finally, Property A2 follows immediately from our definition of $\prec$, the ordering of write operations (well-formedness) and SWA1. Hence, to show that a single writer register implementation satisfies atomicity, it is sufficient to show that it satisfies the properties SWA1-SWA4.

3.2. Time-Complexity of Implementations. We define the time-complexity of atomic register implementation in terms of communication *round-trips*. An operation op invoked by client c consists of a sequence of round-trips, where a round-trip contains the following three phases.

1. Client c sends messages to a subset of processes in a step. (In the first round-trip of the operation, the invocation step of operation precedes this send step.)
2. Upon receiving a message m in step $sp1 =< p, M >$ ($m \in M$), where m is sent by the client c in phase 1 of the round-trip, a process p replies to c 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 $sp2$. (In other words, upon receiving m, p replies to the client before receiving any other messages. Intuitively, this requirement forbids the processes to wait for some other message before replying to m.)
3. Upon receiving a sufficient number of above replies, client c either returns from op or moves to the next round-trip.

We would like to note the following points to emphasize the generality of the above definition: (1) steps from phase 2 and phase 3 of a round-trip may interleave, (2) round-trips of different operations (by different clients) may overlap, and (3) there is no requirement on the communications between a pair of processes both of which are distinct from the client (e.g., the servers may communicate among themselves using any message exchange pattern). Also, note that the above definition is close to the time-complexity definition in [13], where a *round-trip* in the above definition corresponds to a *round* in [13].

3.3. Fast Implementations. A read or a write operation is *fast* if it completes in one communication round-trip. We say that an atomic register implementation is *fast* if both its read and write operations are fast.

Recall that implementations need to tolerate the crash of any client and up to t servers. Hence, in order to ensure termination in a fast implementation, a reader (or a writer) cannot wait for replies from any other client, or more than $S-t$ servers, in the first round-trip of the operation. 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 reply sent by a process to the reader, on receiving a READ message, are of type READACK. Similarly, we define WRITE and WRITEACK messages for fast writes.

We would like to note that no register implementation can have all its write invocations return in the first round-trip before receiving any WRITEACK message. Suppose by contradiction that a write(v) by writer w returns in the first round-trip before receiving any WRITEACK message. Then, w cannot distinguish this operation from another incomplete operation where all WRITE messages of the operation are in transit. In the latter operation, suppose the operation returns before any WRITE message is received, w crashes, and no WRITE messages are ever received. Then, no subsequent read can recover and return v. Thus, in terms of worst-case time-complexity, it is not possible to improve over fast implementations.

4. A Fast Atomic Register Implementation. In the following, we first describe our fast implementation assuming $R < \frac{S}{t} - 2$ and then prove its correctness. For simplicity of presentation, we first present our algorithm assuming that the writer writes timestamps, and the readers read back timestamps. Later we explain how to simply generalize our algorithm such that the writer and the readers associate some value with a timestamp.

4.1. Algorithm. The pseudocode of our fast implementation is given in Figure 4.1. The write procedure is similar to that of [4]. On invoking a write, the writer increments its timestamp (initialized to 0) and sends a WRITE message with the timestamp to all servers (lines 4-5). Upon receiving the message, servers update the timestamp and send WRITEACK messages back to the writer (lines 22-29). The writer returns OK once it has received WRITEACK messages from $S-t$ servers (lines 6-7).

Implementing a fast read is more involved. Like several previous atomic register implementations, 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 line 15, Figure 4.1. However, notice that in our fast implementation server s_i, besides storing the highest received timestamp in its local variable ts_i, maintains also the set $updated_i$ that contains all clients to which s_i has sent the current value of ts_i. Basically, $updated_i$ contains all clients to which s_i has sent an update (in the form of a READACK or a WRITEACK message) about its highest timestamp. This information is read in our read procedure along with the timestamps. Hence, a reader collects timestamps and sets $updated_*$ from $S-t$ servers using READ and READACK messages (lines 12-13 and 32).

Upon receiving $S-t$ READACK messages (collected in the set $rcvMsg$, line 14), the read is precluded from waiting for more messages (the remaining t servers may be faulty). Moreover, in order to have a fast implementation, the read may only perform some local computations and then it must return the value. In our fast implementation, the essence of this local computation is captured by predicate *admissible* (line 8)

```
 0: at the writer w:
 1: procedure initialization:
 2:    ts ← 0
 3: procedure write()
 4:    ts ← ts + 1
 5:    send(WRITE, ts) to all servers
 6:    wait until receive(WRITEACK, ts) from S − t servers
 7:    return(OK)
--------------------------------------------------------------------
    at each reader r_i:
 8: admissible(TS, Msg, a) ≡ ∃μ ⊆ Msg, ∀m ∈ μ :
       (m.ts = TS) ∧ (|μ| ≥ S − at) ∧ (|⋂_{m'∈μ} m'.updated| ≥ a)
 9: procedure initialization:
10:    maxTS ← 0
11: procedure read()
12:    send(READ, maxTS) to all servers
13:    wait until receive(READACK, ∗, ∗) from S − t servers
14:    rcvMsg ← {m|r_i received (READACK, ∗, ∗) in line 13}
15:    maxTS ← Max{ts| (READACK, ts, ∗) ∈ rcvMsg}
16:    if ∃a ∈ [1, R + 1] : admissible(maxTS, rcvMsg, a) then
17:       return(maxTS)
18:    else
19:       return(maxTS − 1)
--------------------------------------------------------------------
    at each server s_i:
20: procedure initialization:
21:    ts_i ← 0; updated_i ← ∅
22: procedure update(ts, c)
23:    if ts > ts_i then
24:       ts_i ← ts; updated_i ← {c}
25:    else
26:       updated_i ← updated_i ∪ {c}
27: upon receive(WRITE, ts) from writer w do
28:    update(ts, w)
29:    send(WRITEACK, ts_i) to w
30: upon receive(READ, ts) from reader r_j do
31:    update(ts, r_j)
32:    send(READACK, ts_i, updated_i) to r_j
```

FIG. 4.1. *Fast SWMR atomic register implementation with* $R < \frac{S}{t} - 2$

which relies on the sets $updated_*$ to evaluate whether the highest received timestamp $maxTS$ may be returned. Namely, if there is an integer a ($1 \leq a \leq R+1$), such that $admissible(maxTS, rcvMsg, a)$ holds in line 16 (we simply say that $maxTS$ is *admissible* (with degree a)), the read returns $maxTS$ (line 17). Otherwise, if $maxTS$ is not admissible, a read returns $maxTS - 1$. In any case, $maxTS$ is cached locally, and is written back by the reader in its following invocation of read (line 12).

In the following, we give an intuition behind the predicate $admissible()$ which is the heart of our fast implementation. The predicate is designed to guarantee that:

(a) $maxTS = k$ is admissible in read rd whenever wr_k precedes rd — this is vital for ensuring "read-write" atomicity, captured by Property SWA2, Section 3.1, and

(b) if $maxTS = k$ is admissible in read rd, then no rd' that follows rd returns a timestamp smaller than k — this is vital for ensuring "read-read" atomicity, captured by Property SWA4, Section 3.1.

First, we explain how our predicate guarantees (a). Consider the following partial run, pr_1. In pr_1, write wr_k ($k \geq 1$) completes by writing k to all servers from some set Σ_1 containing $S-t$ servers. There are no writes in pr_1 that follow wr_k. Moreover, read rd (by some reader r_i), that follows wr_k, reads from set Σ_2 of $S-t$ servers that intersects in $S-2t$ servers with Σ_1, i.e., rd misses t servers in Σ_1. By atomicity, rd must return k in pr_1, and it must do so without waiting for messages from servers from $\overline{\Sigma_2}$ or the writer, since these may be faulty. In this case: (i) in line 15 of rd, $maxTS = k$, and (ii) for every message m received by r_i in rd from servers from $\Sigma_1 \cap \Sigma_2$, we have $m.ts = maxTS$ and $\{w, r_i\} \subseteq m.updated$. Since $|\Sigma_1 \cap \Sigma_2| \geq S-2t$, $maxTS$ is admissible in rd with degree $a = 2$.

On the other hand, the key to guaranteeing (b) is the following invariant (hereafter, $maxTS_{op}$ denotes $maxTS$ computed in line 15 of the read operation op):

LEMMA 1. *Let rd' be a complete read (by reader r_j) that follows a complete read rd (by r_i). If $maxTS_{rd}$ is admissible with degree $a_{rd} \leq R+1$, then:*

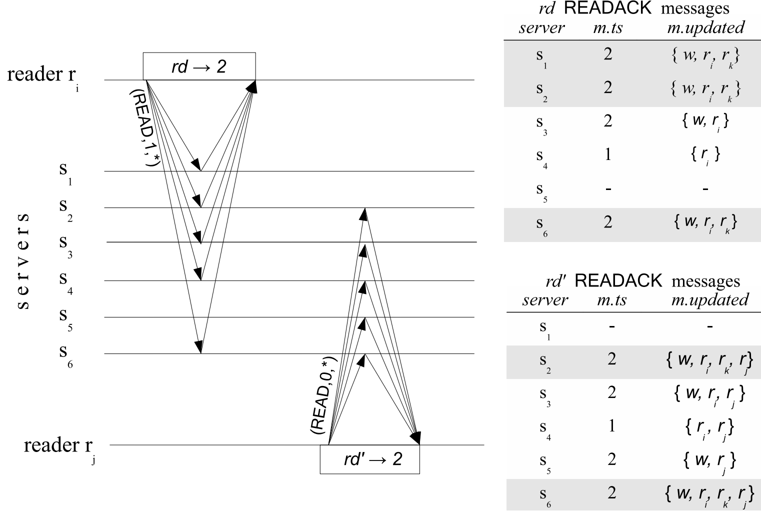
- *$maxTS_{rd'} > maxTS_{rd}$, or*
- *$maxTS_{rd'} = maxTS_{rd}$ and $maxTS_{rd'}$ is admissible with degree $a_{rd}+1$ or degree 1 in rd'.*

Here, we sketch the proof of Lemma 1 (the full correctness proof of our implementation can be found in Section 4.2), and illustrate it in Figure 4.2.

Proof. By the definition of the predicate *admissible* (line 8), there is a set of READACK messages μ_{rd}, sent by servers from the set Σ_{rd} to reader r_i, such that, for every message m in μ_{rd}, $m.ts = maxTS_{rd}$, $|\Sigma_{rd}| = |\mu_{rd}| \geq S - a_{rd}t$ and $|\Pi_{rd}| \geq a_{rd}$, where $\Pi_{rd} = \bigcap_{m \in \mu_{rd}} m.updated$. Notice that, since $a_{rd} \leq R+1$ and $R < \frac{S}{t} - 2$, we have $|\Sigma_{rd}| \geq t+1$. Moreover, since (a) rd' follows rd, (b) $|\Sigma_{rd}| \geq t+1$ and (c) rd' reads from $S-t$ servers, rd' receives a READACK from at least 1 server from Σ_{rd}. Hence, $maxTS_{rd'} \geq maxTS_{rd}$.

If $maxTS_{rd'} = maxTS_{rd} = k$, we distinguish two cases:

Case (i), Fig. 4.2(a), $r_j \notin \Pi_{rd}$ (note that this case is possible only if $a_{rd} \leq R$). It is not difficult to see that k is admissible in rd' with degree $a_{rd}+1$. Indeed, rd' will miss at most t servers from Σ_{rd}, receiving at least $S-(a_{rd}+1)t$ READACK messages containing the timestamp k, and the $updated_*$ fields of these messages will be a su-



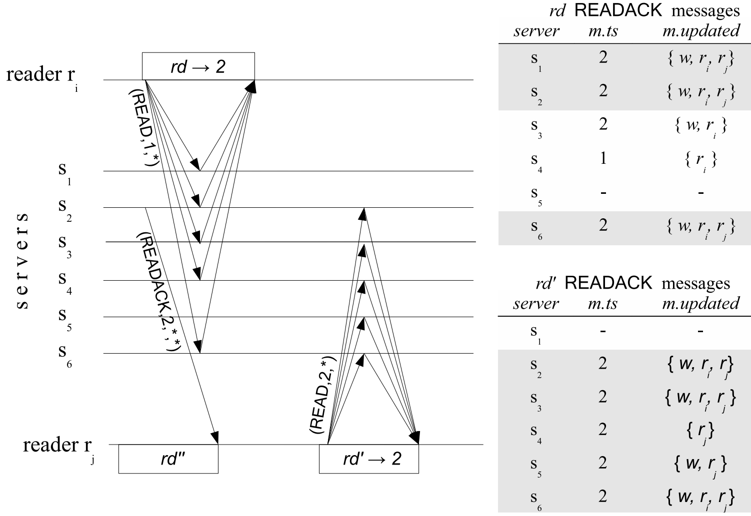
rd READACK messages

server	m.ts	m.updated
s_1	2	$\{w, r_i, r_k\}$
s_2	2	$\{w, r_i, r_k\}$
s_3	2	$\{w, r_i\}$
s_4	1	$\{r_i\}$
s_5	-	-
s_6	2	$\{w, r_i, r_k\}$

rd' READACK messages

server	m.ts	m.updated
s_1	-	-
s_2	2	$\{w, r_i, r_k, r_j\}$
s_3	2	$\{w, r_i, r_j\}$
s_4	1	$\{r_i, r_j\}$
s_5	2	$\{w, r_j\}$
s_6	2	$\{w, r_i, r_k, r_j\}$

(a) In this case $r_j \notin \Pi_{rd} = \bigcap_{m \in \mu_{rd}} m.updated = \{w, r_i, r_k\}$. Since read rd' is guaranteed to READACK messages from at least 2 servers in $\{s_1, s_2, s_6\}$ and these servers add r_j in their *updated* sets, $maxTS_{rd'} = 2$ is admissible with degree 4 in rd' ($\Pi_{rd} = \{w, r_i, r_k, r_j\}$)



rd READACK messages

server	m.ts	m.updated
s_1	2	$\{w, r_i, r_j\}$
s_2	2	$\{w, r_i, r_j\}$
s_3	2	$\{w, r_i\}$
s_4	1	$\{r_i\}$
s_5	-	-
s_6	2	$\{w, r_i, r_j\}$

rd' READACK messages

server	m.ts	m.updated
s_1	-	-
s_2	2	$\{w, r_i, r_j\}$
s_3	2	$\{w, r_i, r_j\}$
s_4	2	$\{r_j\}$
s_5	2	$\{w, r_j\}$
s_6	2	$\{w, r_i, r_j\}$

(b) In this case $r_j \in \Pi_{rd} = \bigcap_{m \in \mu_{rd}} m.updated = \{w, r_i, r_j\}$. Hence, all servers from $\{s_1, s_2, s_6\}$ send a READACK message with $ts = 2$ to r_j *before* rd' is invoked and $maxTS_{rd''} = 2$ in some read rd'' by r_j that precedes rd'. Hence, in rd', r_j sends READ message with $ts = 2$ and $maxTS_{rd'} = 2$ is admissible with degree 1 in rd' ($\Pi_{rd} = \{r_j\}$)

FIG. 4.2. *Illustration of the proof of Lemma 1, for a run with $R = 3$ readers, $S = 6$ servers and $t = 1$.* READACK *messages that allow timestamp 2 to be admissible in* reads *rd and rd' are highlighted in grey. In both cases, in* read *rd, $maxTS_{rd} = 2$ is admissible with degree 3 (notice the* READACK *messages received from servers s_1, s_2 and s_6).*

perset of $\Pi_{rd} \cup \{r_j\}$, hence each containing at least $a_{rd} + 1$ clients.

Case (ii), Fig. 4.2(b), $r_j \in \Pi_{rd}$ (which indicates that rd' is not the first read by r_j). In this case, all servers from Σ_{rd}, at least $t+1$ of them, have sent a READACK message to r_j containing the timestamp k before rd' is invoked. At least one of those must have been received by r_j in read rd'' that immediately precedes rd'. Hence, $maxTS_{rd''} = k$, and $maxTS$ in line 12 of rd' equals k. Finally, $S-t$ servers send READACK to rd' with the timestamp equal to k and the set *updated* that contains $\{r_j\}$ (see lines 22-26), i.e., k is admissible with degree 1 in rd'. □

Finally, to help distinguish READ and READACK messages from different reads of the same reader, we implicitly (for better readability of pseudocode) assume that every reader r_j maintains a local variable $rdCnt_j$ that r_j increments at the beginning of every read invocation. Reader r_j includes $rdCnt_j$ in each READ message (line 12) and servers reply including $rdCnt_j$ in a READACK message (line 32). Servers cache the highest seen $rdCnt_j$ for every reader r_j; namely, before sending a READACK message to r_j, a server stores locally $rdCnt_j$ and does not reply to READ messages from r_j that contain $rdCnt'_j < rdCnt_j$. Hence, once server s_i replies to read rd invoked by reader r_j, s_i replies only to those reads of r_j that follow rd. To refer to this invariant, we say that servers *ignore old reads*.

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.

4.2. Correctness of the Fast Implementation. It is obvious that the read/write time-complexity is one communication round-trip. To show atomicity, we need to prove Properties SWA1-SWA4 of Section 3.1. In the proof, we use the following notation:

DEFINITION 4.1.

1. *$rcvMsg_{op}$ denotes the set of received* READACK *messages the reader collects in* read *operation op (lines 13-14);*
2. *Σ_{op} denotes the set of servers from which the reader received* READACK *messages in $rcvMsg_{op}$ (in case op is a* read*), or the set of servers from which the writer received* WRITEACK *messages in line 6 of op (in case op is a* write*). Notice that, for every operation op, $|\Sigma_{op}| = S - t$;*
3. *$maxTS^{old}_{op}$ denotes the value of $maxTS$ in line 12 in* read *op. Moreover, $maxTS_{op}$ denotes $maxTS$ computed by the reader in line 15, in op (i.e., the highest timestamp in messages in $rcvMsg_{op}$).*
4. *$\mu_{op,a}$ denotes, in case $maxTS_{op}$ is admissible with degree a in op, the subset of $rcvMsg_{op}$, such that (see line 8, definition of predicate admissible):*

(a) $|\mu_{op,a}| \geq S - at$,

(b) $\forall m \in \mu_{op,a}$, $m.ts = maxTS_{op}$, *and*

(c) $|\bigcap_{m\in\mu_{op,a}} m.updated| \geq a$.

5. $\Pi_{op,a}$ *denotes the set of servers* $\bigcap_{m\in\mu_{op,a}} m.updated$; *and*
6. *Finally,* $\Sigma_{\mu_{op,a}}$ *denotes the set of servers that sent messages in* $\mu_{op,a}$.

We start with two simple observations that we use in the rest of the proof.

LEMMA 4.2. *If a server sets its local variable ts to x at time T, then the server never sets ts to a value that is lower than x after time T.*

Proof. By trivial server code inspection. □

LEMMA 4.3. *A* read *operation rd may only return either $maxTS_{rd}$ or $maxTS_{rd}-1$.*

Proof. By lines 16-19 and the definition of $maxTS_{rd}$. □

The following lemma captures the "writeback" mechanism of our algorithm and states, roughly, that read operations invoked by the same reader cannot violate atomicity. This lemma is crucial for proving Properties SWA1 and SWA2.

LEMMA 4.4. *A* read *rd cannot return a value smaller than $maxTS^{old}_{rd}$.*

Proof. Recall that $maxTS^{old}_{rd}$ denotes $maxTS$ sent in READ message in rd (line 12). By lines 23-24, every READACK message received by a reader in rd from some server s_j is with $ts_j \geq maxTS^{old}_{rd}$. Hence, $maxTS_{rd} \geq maxTS^{old}_{rd}$. There are the following two cases to consider. (1) If $maxTS_{rd} > maxTS^{old}_{rd}$, by Lemma 4.3, the return value is not smaller than $maxTS^{old}_{rd}$. (2) If $maxTS_{rd} = maxTS^{old}_{rd}$. then every READACK message in $rcvMsg_{rd}$ has a timestamp equal to $maxTS^{old}_{rd}$ and $r_i \in updated$. Since rd receives $S-t$ READACK messages, $maxTS^{old}_{rd}$ is admissible with degree $a=1$ ($\mu_{rd,1} = rcvMsg_{rd}$). By lines 16-17, rd returns $maxTS^{old}_{rd}$. □

LEMMA 4.5. ***(SWA1)*** *If a* read *returns, it returns a non-negative timestamp.*

Proof. To prove the lemma, it is sufficient to show that there is no read rd in which $maxTS^{old}_{rd} < 0$ (then, the lemma follows from Lemma 4.4).

To see this, assume by contradiction that there is a read rd by reader r_i in which $maxTS^{old}_{rd} < 0$. By lines 9-10, this is not the first read by r_i, i.e., there is a read rd' by r_i that (immediately) precedes r_i such that $maxTS_{rd'} < 0$, i.e., $S-t$ servers sent READACK message in rd' with $ts_* < 0$. However, since server timestamps are initialized to 0, this contradicts Lemma 4.2. □

LEMMA 4.6. ***"read-write" atomicity (SWA2).*** *If a* read *rd returns timestamp l and rd follows* write *wr_k, then $l \geq k$.*

Proof. Denote by r_i the reader that invoked rd and let $\Sigma' = \Sigma_{wr_k} \cap \Sigma_{rd}$. Since $|\Sigma_{wr_k}| = S-t$ and $|\Sigma_{rd}| = S-t$, we have $|\Sigma'| \geq S-2t$.

When a server s_j in Σ_{wr_k} (and, hence, in Σ') replies to a WRITE message from wr_k, its ts_j is at least k (server timestamp is not smaller than k due to the condition in line 23). Since wr_k precedes rd, by Lemma 4.2, servers in Σ' reply with $ts_* \geq k$ to rd. Hence, $maxTS_{rd} \geq k$. There are the following two cases to consider:

1. $maxTS_{rd} > k$
   By Lemma 4.3, rd does not return a timestamp lower than k.
2. $maxTS_{rd} = k$
   Let μ' be the set of READACK messages sent by servers in Σ' to rd. By definition of Σ_{rd} and since $\Sigma' \subseteq \Sigma_{rd}$, we have $\mu' \subseteq rcvMsg_{rd}$. Since (a) every server $s_j \in \Sigma'$ replies to rd with $ts_j \geq k$, (b) $\mu' \subseteq rcvMsg_{rd}$ and (c)

$maxTS_{rd} = k$, we have that every server $s_j \in \Sigma'$ replies with $ts_j = k$ to rd (and r_j receives these replies).
Moreover, since every server $s_j \in \Sigma'$ replies $ts_j = k$ to wr_k (since $\Sigma' \subseteq \Sigma_{wr_k}$) before sending $ts_j = k$ to rd (since wr_k precedes rd), for every message m in μ', $w \in m.updated$. Furthermore, since s_j replies with $ts_j = k$ to rd, by line 26, $r_i \in m.updated$. Thus, $\{w, r_i\} \subseteq \bigcap_{m \in \mu'} m.updated$. As $|\Sigma'| \geq S - 2t$, $maxTS_{rd}$ is admissible in rd with degree $a = 2$. Hence, rd returns $maxTS_{rd} = k$.

□

The following lemma helps prove Property SWA3.

LEMMA 4.7. *If $maxTS_{rd} \geq k$, then rd does not precede wr_k.*

Proof. We focus on the case $k \geq 1$, since the proof for $k = 0$ follows from the definition of wr_0. To prove the lemma, it is sufficient to show that no server sets its local timestamp to a value greater or equal to k, before wr_k is invoked. Assume, by contradiction, that there is such server s_i that is, moreover, the first server to set its local timestamp ts_i to $l \geq k$ according to the global clock (at time T), i.e., no server sets its local timestamp to l before time T.

It is obvious that wr_k and wr_l (which might be the same operations) are invoked after T. Hence, s_i must have set ts_i to l after receiving a READ message in a read rd' invoked by reader r_j in which $maxTS^{old}_{rd'} = l$. Since $l \geq k \geq 1$, there is a read rd'' by r_j that immediately precedes rd' in which $maxTS_{rd''} = l$. Since rd'' precedes rd', rd'' completes before time T. By definition of $maxTS_{rd''}$, some server had set its local timestamp to l before rd'' completed. A contradiction with the assumption that no server sets its local timestamp before time T. □

Lemma 4.7 has the following important corollary.

COROLLARY 4.8. *If $maxTS_{rd} = k \geq 1$ then write wr_{k-1} completes before read rd completes.*

LEMMA 4.9. ***(SWA3)*** *If a read rd returns timestamp k ($k \geq 1$), then rd does not precede wr_k.*

Proof. By Lemmas 4.3 and 4.7. □

We now proceed towards proving Property SWA4 of Section 3.1 ("read-read" atomicity). The following 2 auxiliary lemmas are related to the predicate *admissible* and to the sizes of the relevant subsets of the set $rcvMsg$.

LEMMA 4.10. *If $maxTS_{rd}$ is admissible in rd with degree a, then $\Sigma_{\mu_{rd,a}}$ contains at least t+1 servers.*

Proof. By Definition 4.1 and inequalities $a \leq R + 1$ and $R < \frac{S}{t} - 2$, we have $|\Sigma_{\mu_{rd,a}}| \geq S - at > (R + 2)t - (R + 1)t = t$.□

LEMMA 4.11. *Assume that $maxTS_{rd}$ is admissible with degree $a \in [1, R + 1]$ in some read rd and that a complete read rd' follows rd. Then, the number of servers in $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ is at least $S - (a + 1)t$. Moreover, $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ contains at least one server.*

Proof. Since $|\Sigma_{\mu_{rd,a}}| = |\mu_{rd,a}| \geq S - at$ and $|\Sigma_{rd'}| = S - t$, it follows that $|\Sigma_{rd'} \cap \Sigma_{\mu_{rd,a}}| \geq S - (a + 1)t$. Moreover, since $a \in [1, R + 1]$ and $t < S/(R + 2)$, we have $S - (a + 1)t \geq 1$. □

The following Lemma proves the key invariant used in the proof of Property SWA4 of Section 3.1.

LEMMA 4.12. *Assume that:*

1. *$maxTS_{rd}$ is admissible with degree $a \in [1, R + 1]$ in some read rd,*

2. *a complete* read rd' *by reader* r_j *follows* rd,

3. *there is a set* $X \subseteq \Sigma_{\mu_{rd,a}}$ *of at least* $t+1$ *servers, such that for all* $s_i \in X$, s_i *sends message* $m_i \in \mu_{rd,a}$ *with* $r_j \in m_i.updated$.

Then, rd' *does not return a value smaller than* $maxTS_{rd}$.

Proof. Since the messages in $\mu_{rd,a}$ are sent before the completion of rd (and hence, before the invocation of rd') and since, there is a server s_i that sends $m_i \in \mu_{rd,a}$ with $r_j \in m_i.updated$, r_j has invoked at least one read before rd'. Let rd'' be the last read by reader r_j which precedes rd'. Since $|X| \geq t+1$ and $|\Sigma_{rd''}| = S-t$, there is at least one server s_k in $X \cap \Sigma_{rd''}$, such that the READACK message m sent by s_k is received by r_j in rd''. In the following paragraph, we show that $m.ts \geq maxTS_{rd}$.

By contradiction, assume $m.ts < maxTS_{rd}$. Since servers ignore old reads,there is a read rd_α by r_j, such that rd_α follows rd'' and s_k sends a READACK message m_α to rd_α, before s_k sends $m_k \in \mu_{rd,a}$, i.e., before rd' is invoked. Hence, rd'' is not the last read by reader r_j which precedes rd'. A contradiction.

Since $m.ts \geq maxTS_{rd}$ and $m \in rcvMsg_{rd''}$, we have $maxTS_{rd''} \geq maxTS_{rd}$. Since rd' follows rd'', it follows that r_j in rd' sends READ messages with $ts \geq maxTS_{rd}$. By Lemma 4.4, rd' returns a timestamp greater than or equal to $maxTS_{rd}$. □

LEMMA 4.13. ***"read-read" atomicity (SWA4).*** *If* reads rd_1 *and* rd_2 *return timestamps* ret_{rd_1} *and* ret_{rd_2}, *respectively, and if* rd_2 *follows* rd_1, *then* $ret_{rd_2} \geq ret_{rd_1}$.

Proof. Without loss of generality we can assume (to facilitate the notation) that read rd_1 (resp., rd_2) is invoked by reader r_1 (resp., r_2). Suppose $r_1 = r_2$. Then, in the read that immediately follows rd_1, r_1 sends a READ message with $ts \geq ret_{rd_1}$, and hence, by Lemma 4.4, the read returns a value greater than or equal to ret_{rd_1}. Using Lemma 4.4 and a simple induction, we can derive that any read by r_1 which follows rd_1 (including rd_2) returns $ts \geq ret_{rd_1}$. Therefore, in the rest of the proof we assume that $r_1 \neq r_2$. We distinguish the following two cases:

1. $maxTS_{rd_1}$ is not admissible in rd_1.
   It follows that $ret_{rd_1} = maxTS_{rd_1} - 1$. By Lemma 4.5, $ret_{rd_1} \geq 0$ and, hence, $maxTS_{rd_1} \geq 1$. By Corollary 4.8, $wr_{ret_{rd_1}}$ completes before rd_1 is completes. Since rd_1 precedes rd_2, it follows that $wr_{ret_{rd_1}}$ precedes rd_2. By Lemma 4.6, rd_2 returns $ret_{rd_2} \geq ret_{rd_1}$.
2. $maxTS_{rd_1}$ is admissible in rd_1.
   It follows that $ret_{rd_1} = maxTS_{rd_1}$, and there is some $a \in [1, R+1]$ such that ret_{rd_1} is admissible in rd_1 with degree a. By Lemma 4.11, there is at least one server $s_i \in \Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$. Since rd_1 precedes rd_2, s_i first replies with $ts_i = ret_{rd_1}$ to rd_1 before s_i replies to rd_2. Finally, by Lemma 4.2, it follows that s_i replies to rd_2 with $ts_i \geq ret_{rd_1}$, i.e., $maxTS_{rd_2} \geq ret_{rd_1}$.
   We distinguish the following three exhaustive cases:
   (a) $maxTS_{rd_2} > ret_{rd_1}$
   By Lemma 4.3, we have $ret_{rd_2} \geq ret_{rd_1}$.
   (b) $maxTS_{rd_2} = ret_{rd_1}$ and $maxTS_{rd_2}$ is admissible in rd_2
   By lines 16-17, $ret_{rd_2} = maxTS_{rd_2} = ret_{rd_1}$.
   (c) $maxTS_{rd_2} = ret_{rd_1}$ and $maxTS_{rd_2}$ is not admissible in rd_2
   We show that this case is impossible by exhibiting appropriate contradictions.
   In this case, by lines 16-19, $ret_{rd_2} = maxTS_{rd_2} - 1 = ret_{rd_1} - 1 =$

$maxTS_{rd_1} - 1$. By Lemma 4.11, there is at least one server in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$. Since rd_1 precedes rd_2 and servers in $\Sigma_{\mu_{rd_1,a}}$ reply with $ts_* = ret_{rd_1}$ to rd_1, by Lemma 4.2, servers in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$ reply to rd_2 with $ts_* \geq ret_{rd_1}$. Since $ret_{rd_2} + 1 = ret_{rd_1} = maxTS_{rd_2}$, every server in $\Sigma_{rd_2} \cap \Sigma_{\mu_{rd_1,a}}$ replies to rd_2 with $ts = ret_{rd_1} = ret_{rd_2} + 1$. There are the following two cases to consider:

i. $a \leq R$
   In this case, by Lemma 4.11, the number of servers in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$ is greater than t. Let μ_1 be the set of READACK messages sent from servers in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$ to rd_1. There are two cases to consider:

   A. $r_2 \notin \bigcap_{m \in \mu_1} m.updated$
      Denote $\bigcap_{m \in \mu_1} m.updated$ by Π_1. Notice that, by definitions of μ_1 and $\mu_{rd_1,a}$, $\mu_1 \subseteq \mu_{rd_1,a}$. Hence, we have $\Pi_1 \supseteq \Pi_{rd_1,a}$ and $|\Pi_1| \geq a$.
      Let μ_2 be the set of messages received by rd_2 from servers in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$. For any server $s_i \in \Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$, let m_1 and m_2 be the messages sent by s_i in μ_1 and μ_2 respectively. Since we know that $m_1.ts_i = m_2.ts_i = ret_{rd_1}$ and since m_1 is sent before m_2, we have $m_1.updated \subseteq m_2.updated$. Hence, $\Pi_1 \subseteq \bigcap_{m \in \mu_2} m.updated$. Since every server which replies to r_2 in rd_2, adds r_2 to its *updated* set before replying to r_2, $r_2 \in \bigcap_{m \in \mu_2} m.updated$.
      Since $r_2 \notin \Pi_1$, $|\bigcap_{m \in \mu_2} m.updated| \geq |Pi_1| + 1 \geq a + 1$. Since (a) the number of messages in μ_2 equals the number of servers in $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$ and (b) $a + 1 \leq R + 1$, by Lemma 4.11 and the definition of predicate *admissible* (line 8), we have that $ret_{rd_1} = ret_{rd_2} + 1$ is admissible in rd_2 with degree $a + 1$. Hence, the timestamp returned by rd_2 is $ret_{rd_2} + 1$, a contradiction (with the assumption that rd_2 returns ret_{rd_2}).

   B. $r_2 \in \bigcap_{m \in \mu_1} m.updated$
      Denote by X the set $\Sigma_{\mu_{rd_1,a}} \cap \Sigma_{rd_2}$. By definition of μ_1, messages in μ_1 are sent by processes in X. By Lemma 4.11 and since $a \leq R$, it follows that the number of servers in X is greater than t. Hence, by Lemma 4.12, $ret_{rd_2} \geq maxTS_{rd_1}$. A contradiction with $ret_{rd_2} = maxTS_{rd_1} - 1$.

ii. $a = R + 1$
   Since $|\{w, r_1, ..., r_R\}| = R + 1$ and $|\bigcap_{m \in \mu_{rd_1,a}} m.updated| \geq a = R + 1$, we have $r_2 \in \bigcap_{m \in \mu_{rd_1,a}} m.updated$. By Lemma 4.10, $\Sigma_{\mu_{rd_1,a}}$ contains at least $t+1$ servers. Replacing X with $\Sigma_{\mu_{rd_1,a}}$ in Lemma 4.12, it follows that $ret_{rd_2} \geq maxTS_{rd_1}$. A contradiction with $ret_{rd_2} = maxTS_{rd_1} - 1$.

□

Finally, we combine the above lemmas to prove the correctness of our fast implementation.

THEOREM 4.14 (Fast atomic register). *The algorithm of Figure 4.1 is a fast implementation of an atomic SWMR register.*

Proof. Atomicity follows from Lemmas 4.5, 4.6, 4.9 and 4.13. Moreover, it is

obvious that our implementation satisfies Termination (conditions in lines 6 and 13 are non-blocking). Finally, our implementation is fast: all operations involve a single communication round-trip between a client and servers.□

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

PROPOSITION 5.1. *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 that 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.

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., $R > S-2$).

Notice that our model does not require that a message sent by a faulty process is received by the receiver. Hence, in any (partial) run, it is possible any message sent by a faulty process remains in transit (i.e., that such a message is never received by the receiver). In our proof we construct (partial) runs in which, unless explicitly stated otherwise, all messages sent by faulty processes are in transit.

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 these 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 these reply messages and returns from the invocation.*

Block diagrams. To depict our proof, we use block diagrams (see Fig. 5.1). We depict an operation *op* through a set of rectangles, (generally) arranged in a single column. In the column corresponding to some operation *op*, we draw a rectangle in the ith row, if all servers in block B_i have received the READ or WRITE message from *op* and have sent reply messages, i.e., we draw a rectangle in the ith row if *op* does not skip B_i.

We illustrate a particular instance of the proof in Figure 5.2 and Figure 5.3, where $R = 3$ and the set of servers are partitioned into five blocks, B_1 to B_5.

We now proceed to the proof of Proposition 5.1.

Proof. 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 operation skips B_{R+2}. We define a series of partial runs each of which can be

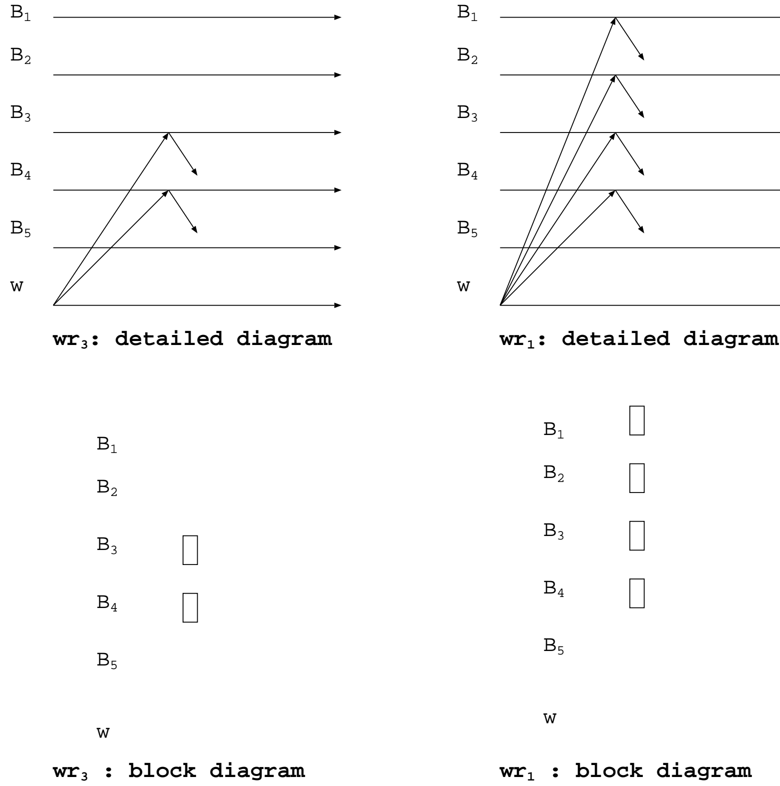
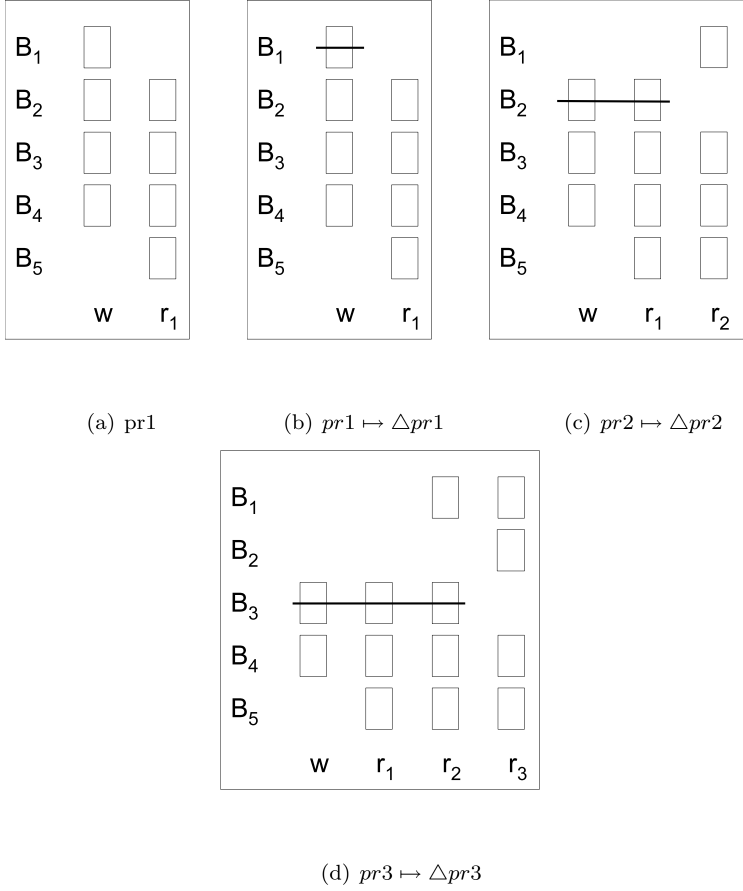


FIG. 5.1. *Block diagrams*

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) 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 the write completes, and hence, (3) wr and wr_1 differ only at w. A sample partial writes, wr_1 and wr_3 are presented in Figure 5.1.

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 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



(a) pr1 (b) $pr1 \mapsto \triangle pr1$ (c) $pr2 \mapsto \triangle pr2$

(d) $pr3 \mapsto \triangle pr3$

Fig. 5.2. *Partial runs: pr_i (including the crossed out rectangles) and $\triangle pr_i$ (excluding the crossed out rectangles)*

reads one after the other: for $1 \leq h \leq i$, r_h does a read that skips $\{B_j | h \leq j \leq i\}$. Here, the first $i-1$ appended reads are incomplete whereas the last one is complete. Figure 5.2 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 $\{B_j | h \leq j \leq R\}$. 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) 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

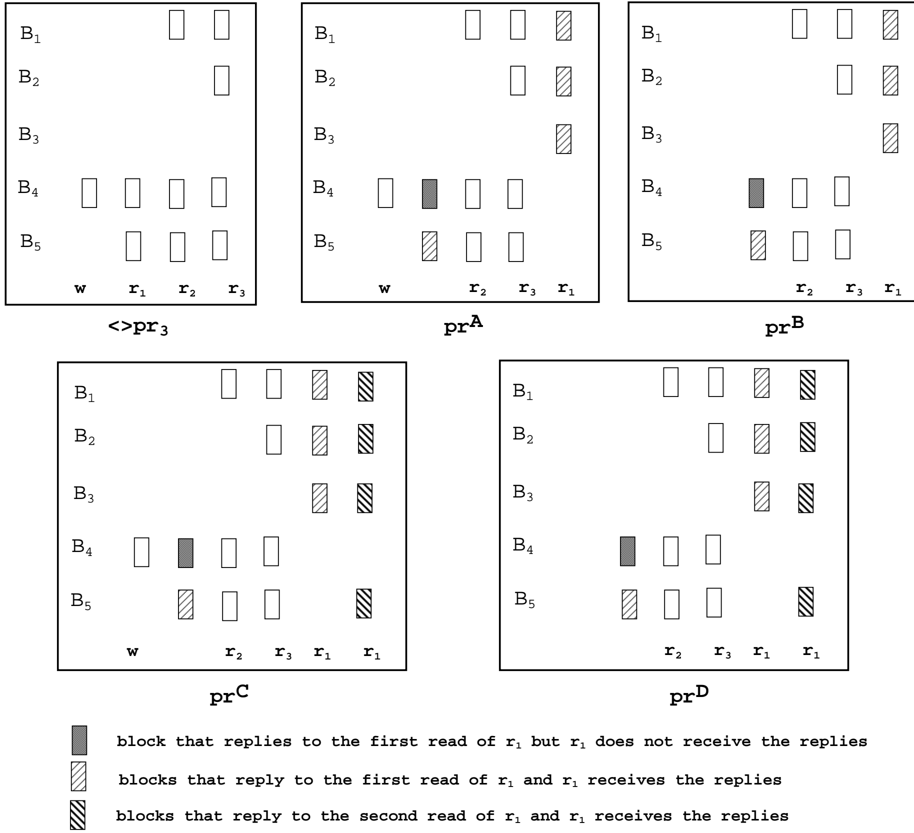


FIG. 5.3. *Partial runs:* pr^A*,* pr^B*,* pr^C *and* pr^D

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 5.3 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 5.3). 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 5.3). 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 follows r_R's read which returns 1. Clearly, partial run pr^C violates atomicity. □

6. Arbitrary Failure Model. In this section we consider fast implementations that tolerate arbitrary failures of servers and readers, but the writer can fail only by crashing (intuitively, in our single-writer setting, arbitrarily faulty writer could make the shared data structure useless). 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 allow for *any* number of arbitrarily faulty readers and distinguish two resilience thresholds for server failures: b and t [19]. Just as in the crash-stop model, a maximum number of t servers can fail. However, out of these t servers, at most b can be malicious. We therefore always have $b \leq t$. In the literature, the special case where $b = t$ is usually considered. However, by considering b and t separately we can directly generalize the results in the previous sections (i.e. with $b = 0$).

In our arbitrary failure model, malicious processes an deviate arbitrarily from automata assigned to them, with the restriction with respect to creation of digital signatures of other processes as detailed below. In the remainder of this Section, we say that a process is faulty if it fails by crashing or if it is malicious (otherwise, a process is called correct). We also say that a process is non-malicious if it is correct or fails by crashing.

In this paper, we assume that a process can produce cryptographic digital signatures (e.g., [28]). The functionality of the digital signature scheme provides two operations: σ for signing and Ver for signature verification. The invocation of σ takes a process ID, say p and a bit string m as parameters and returns a bit string sig, called *signature*. The verification operation Ver takes a process ID p, and two bit strings m and sig as parameters and returns a boolean. The verification function has the property that $Ver(p, m, sig)$ invoked by a benign process evaluates to *true* if and only if process p executed $\sigma(p, m)$ in some previous step. Furthermore, no process (including Byzantine ones) other than p may invoke $\sigma(p, m)$ (we say signatures are *unforgeable*); hence, alternatively, we also write $\sigma(p, m)$ as $\sigma_p(m)$.

Finally, given that we allow for arbitrarily faulty readers, we require atomicity to hold in a given run only on the subset of non-malicious read/write operations invoked by non-malicious clients (i.e., when read invocations/responses at malicious readers are removed from a run). For simplicity of presentation, we refer to an operation invoked by a non-malicious client as to a *non-malicious operation.*

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 6.1). 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 it sends to servers. Apart from the addition of digital signatures, the write mechanism is unchanged and the writer waits for the response of $S - t$ servers. Server code related

---

0: **at the writer** w**:**
1: **procedure** initialization:
2: $\quad ts \leftarrow 0$
3: **procedure** write()
4: $\quad ts \leftarrow ts + 1$
5: $\quad$ send (WRITE, ts, $\sigma_w(ts)$) to all servers
6: $\quad$ **wait until** receive (WRITEACK, ts) from $S - t$ servers
7: $\quad$ return(OK)

---

at each reader r_i**:**
8: $admissible(TS, Msg, a) \equiv \exists \mu \subseteq Msg, \forall m \in \mu :$
$\quad (m.ts = TS) \wedge (|\mu| \geq S - at - (a-1)b) \wedge (|\bigcap_{m' \in \mu} m'.updated| \geq a)$

9: **procedure** initialization:
10: $\quad maxTS \leftarrow 0$; $sig \leftarrow \perp$
11: **procedure** read()
12: $\quad$ send(READ, $maxTS$, sig) to all servers
13: $\quad$ **wait until** receive (READACK, ts', sig', $updated'$) from $S - t$ servers, such that:
$\quad\quad ts' \geq maxTS$ **and** $r_i \in updated'$ **and** $Ver(w, ts', sig')$
14: $\quad rcvMsg \leftarrow \{m | r_i$ received (READACK, $*$, $*$, $*$) in line 13 $\}$
15: $\quad maxTS \leftarrow \mathbf{Max}\{ts|$ (READACK, ts, $*$, $*$) $\in rcvMsg\}$
16: $\quad sig \leftarrow sig_{maxTS}$: (READACK, $maxTS$, sig_{maxTS}, $*$) $\in rcvMsg$
17: $\quad$ **if** there is $a \in [1, R+1]$: $admissible(maxTS, rcvMsg, a)$ **then**
18: $\quad\quad$ return($maxTS$)
19: $\quad$ **else**
20: $\quad\quad$ return($maxTS - 1$)

---

at each server s_i**:**
21: **procedure** initialization:
22: $\quad ts_i \leftarrow 0$; $updated_i \leftarrow \emptyset$; $sig_i \leftarrow \perp$
23: **procedure update**(ts, sig, c)
24: $\quad$ **if** $ts > ts_i$ **then**
25: $\quad\quad ts_i \leftarrow ts$; $sig_i \leftarrow sig$; $updated_i \leftarrow \{c\}$
26: $\quad$ **else**
27: $\quad\quad updated_i \leftarrow updated_i \cup \{c\}$
28: **upon** receive(WRITE, ts, sig) from writer w **and** $Ver(w, ts, sig)$ **do**
29: $\quad$ **update**(ts, sig, w)
30: $\quad$ send(WRITEACK, ts_i) to w
31: **upon** receive(READ, ts, sig) from reader r_j **and** $Ver(w, ts, sig)$ **do**
32: $\quad$ **update**(ts, sig, r_j)
33: $\quad$ send(READACK, ts_i, sig_i, $updated_i$) to r_j

FIG. 6.1. *Fast atomic storage implementation with* $S \geq (R+2)t + (R+1)b + 1$

to write operation is also unchanged, except that servers store the digital signature of every value they store in addition to the value itself.

Our read procedure begins with servers issuing a READ message containing the highest timestamp encountered in the previous read invocation (line 12) along with the respective signature of the writer. In a way, the reader writes back this timestamp, signed by the writer, 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 known to all readers (and hence, needs not be digitally signed by the writer). Then, the reader collects responses from $S - t$ servers containing the latest timestamps encountered by the servers (including the one being written back by the reader); all the timestamps need to be accompanied with the verifiable signature of the writer (line 13). The reader then selects the highest such timestamp, $maxTS$ (line 15). Moreover, the reader stores the corresponding writer's signature into variable sig (line 16). The pair $(maxTS, sig)$ will be written back by the reader in its next read invocation (lines 12, 23-27 and 31-33).

Apart from using digital signatures as described above, the mechanism of the read procedure is very similar to our crash-tolerant algorithm of Figure 4.1 (Sec. 4.1). The additional difference is related to predicate *admissible* in line 8, which checks if the latest value has been seen by a sufficient number of servers, and which is slightly modified. Namely, in order for $maxTS$ (i.e., the highest timestamp received in a read) to be admissible with degree a, $maxTS$ must have been reported to the reader by at least $S - at - (a-1)b$ servers. This is to be contrasted with at least $S - at$ reports needed in the crash-only case (notice here that the number of servers S is not identical in the crash-only and the arbitrary failure cases, being higher in the latter). To see why our algorithm requires $S - at - (a-1)b$ confirmations from different servers, 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; denote this set of servers by S_1. Subsequently, a reader reads from a set S_2 (of $S-t$ servers) that overlaps in $S-2t-b$ (non-malicious) servers with S_1, i.e., the reader 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 $S_1 \cap S_2$. 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 $S_1 \cap S_2$, 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. This can be extrapolated further depending on the number of the readers in the system. Hence the need for as few as $S - at - (a-1)b$ different confirmations for a timestamp to be admissible (with degree a).

We now prove the correctness of the fast implementation depicted in Figure 6.1.

6.2. Correctness of the Fast Implementation. The skeleton of the proof follows the proof of our crash-tolerant algorithm (Sec. 4.2); differences in two proofs account for counteracting possible actions of malicious processes, the use of digital signatures and modifications of predicate *admissible*. In the following, we omit correctness proofs of those lemmas that can be trivially obtained from the proof of their counterparts from Section 4.2, by inserting the attribute "non-malicious" before every occurrence of "server", "reader" and "read" and by replacing every reference to

Lemma/Corollary 4.x, by Lemma/Corollary 6.x. However, for completeness, we repeat the statement of each of the lemmas in the arbitrary failure context. In our proof, we maintain the assumption that non-malicious servers ignore old reads as explained in Sec. 4.1.

First we modify some notation of the Definition 4.1 as follows (other notation from Def. 4.1 remains):

DEFINITION 6.1.

- $\mu_{op,a}$ *denotes, in case* $maxTS_{op}$ *is admissible with degree* a *in* op*, the subset of* $rcvMsg_{op}$*, such that (see line 8, Fig. 6.1):*

  (a) $|\mu_{op,a}| \geq S - at - (a-1)b$,

  (b) $\forall m \in \mu_{op,a}$, $m.ts = maxTS_{op}$ *and*

  (c) $|\bigcap_{m \in \mu_{op,a}} m.updated| \geq a$.

LEMMA 6.2. *If a non-malicious server* s_i *sets its local variable* ts_i *to* x *at time* T*, then* s_i *never sets* ts_i *to a value that is lower than* x *after time* T*.*

LEMMA 6.3. *A* read *operation* rd *may only return either* $maxTS_{rd}$ *or* $maxTS_{rd}-1$.

LEMMA 6.4. *A non-malicious* read rd *cannot return a value smaller than* $maxTS_{rd}^{old}$*.*

Proof. By line 24, every READACK message received by a non-malicious reader in rd from a non-malicious server is with a timestamp at least $maxTS_{rd}^{old}$. The reader awaits for $S-t$ READACK messages before returning a value. Moreover, reader discards all READACK messages that have a timestamp less than $maxTS_{rd}^{old}$ (line 13), as those READACK messages are clearly from malicious servers. Eventually, since we assume at most t server failures, rd receives READACK messages from $S-t$ non-malicious servers that satisfy conditions in line 13. Clearly, $maxTS_{rd} \geq maxTS_{rd}^{old}$. There are the following two cases to consider. (1) If $maxTS_{rd} > maxTS_{rd}^{old}$, then, by Lemma 6.3, the return value is not smaller than $maxTS_{rd}^{old}$. (2) If $maxTS_{rd} = maxTS_{rd}^{old}$, then every READACK message in $rcvMsg_{rd}$, at least $S-t$ of them, has timestamp equal to $maxTS_{rd}^{old}$ and has $r_i \in updated$ (possibly different READACK messages from malicious servers are discarded in line 13). Hence, predicate *admissible* holds for $maxTS_{rd}$ with degree $a = 1$ and rd returns $maxTS_{rd}^{old}$. □

LEMMA 6.5. ***(SWA1)*** *If a non-malicious* read *returns, it returns a non-negative timestamp.*

Proof. To prove the lemma, it is sufficient to show that there is no non-malicious read rd in which $maxTS_{rd}^{old} < 0$ (then, the lemma follows from Lemma 4.4).

To see this, assume by contradiction that there is a read rd by non-malicious reader r_i in which $maxTS_{rd}^{old} < 0$. Moreover, without loss of generality, we can fix rd such that there is no read rd' by r_i such that rd' precedes rd and $maxTS_{rd'}^{old} < 0$. By lines 9-10, this rd is not the first read by r_i, i.e., there is a read rd' by r_i that (immediately) precedes r_i such that $maxTS_{rd'} < 0$ and $maxTS_{rd'}^{old} \geq 0$. However, this contradicts the condition in line 13 that requires $maxTS_{rd'} \geq maxTS_{rd'}^{old}$. □

LEMMA 6.6. ***"read-write" atomicity (SWA2).*** *If a non-malicious* read rd *returns timestamp* l *and* rd *follows* write wr_k*, then* $l \geq k$*.*

Proof. Denote by r_i the non-malicious reader that invoked rd and let $\Sigma' = \Sigma_{wr_k} \cap \Sigma_{rd}$ and let Σ^{NM} be the subset of Σ' that contains only non-malicious (NM) servers. Obviously, $|\Sigma^{NM}| \geq S - 2t - b$.

When a non-malicious server s_j in Σ_{wr_k} (and, hence, in Σ^{NM}) replies to a WRITE message from wr_k, its ts_j is at least k (the timestamp is not smaller than k due to the condition in line 24). Since wr_k precedes rd, by Lemma 6.2, servers in Σ' reply with $ts_* \geq k$ to rd. Hence, $maxTS_{rd} \geq k$. There are the following two cases to consider:

1. $maxTS_{rd} > k$
   By Lemma 6.3, rd does not return a timestamp lower than k.
2. $maxTS_{rd} = k$
   Let μ^{NM} be the set of READACK messages sent by servers in Σ^{NM} to rd. By definition of Σ_{rd} and since $\Sigma^{NM} \subseteq \Sigma_{rd}$, we have $\mu^{NM} \subseteq rcvMsg_{rd}$. Since (a) every server $s_j \in \Sigma^{NM}$ replies to rd with $ts_j \geq k$, (b) $\mu^{NM} \subseteq rcvMsg_{rd}$ and (c) $maxTS_{rd} = k$, we have that every server $s_j \in \Sigma^{NM}$ replies with $ts_j = k$ to rd (and r_j receives these replies).
   Moreover, since every server $s_j \in \Sigma^{NM}$ replies $ts = k$ to wr_k (since $\Sigma^{NM} \subseteq \Sigma_{wr_k}$) before sending $ts = k$ to rd (since wr_k precedes rd), for every message m in μ^{NM}, $w \in m.updated$. Furthermore, since s_j replies with $ts_j = k$ to rd, by line 27, $r_i \in m.updated$. Thus, $\{w, r_i\} \subseteq \bigcap_{m\in\mu'} m.updated$. As $|\Sigma^{NM}| \geq S - 2t - b$, $maxTS_{rd}$ is admissible in rd with degree $a = 2$. Hence, rd returns $maxTS_{rd} = k$.

□

LEMMA 6.7. *If $maxTS_{rd} \geq k$ in a non-malicious* read *rd, then rd does not precede wr_k.*

Proof. In case $k = 0$, the lemma follows directly from the definition of wr_0. In case $k \geq 1$, the proof follows directly from the unforgeability of writer's signatures and the fact that the writer does not sign any value greater or equal to k before it invokes wr_k. Hence, no timestamp $k' \geq k$ can pass the signature verification check in line 13 before wr_k is invoked. □

COROLLARY 6.8. *If $maxTS_{rd} = k \geq 1$ in a non-malicious* read*, then* write *wr_{k-1} completes before* read *rd completes.*

LEMMA 6.9. ***(SWA3)*** *If a non-malicious* read *rd returns timestamp k ($k \geq 1$), then rd does not precede wr_k.*

LEMMA 6.10. *If $maxTS_{rd}$ is admissible with degree a in non-malicious* read *rd, then $\Sigma_{\mu_{rd,a}}$ contains at least t+1 non-malicious servers.*

Proof. By Definitions 4.1 and 6.1, and inequalities $a \leq R+1$ and $R < \frac{S+b}{t+b} - 2$, we have $|\Sigma_{\mu_{rd,a}}| \geq S - at - (a-1)b > (R+2)t + (R+1)b - (R+1)t - Rb > t + b$. Since we assume at most b malicious servers, $\Sigma_{\mu_{rd,a}}$ contains at least t+1 non-malicious servers. □

LEMMA 6.11. *Assume that $maxTS_{rd}$ is admissible with degree $a \in [1, R+1]$ in some non-malicious* read *rd and that a complete non-malicious* read *rd' follows rd. Then, the number of non-malicious servers in $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ is at least $S-(a+1)t-ab$. Moreover, $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ contains at least one non-malicious server.*

Proof. Since $|\Sigma_{\mu_{rd,a}}| = |\mu_{rd,a}| \geq S - at - (a-1)b$ and $|\Sigma_{rd'}| = S - t$, it follows that $|\Sigma_{rd'} \cap \Sigma_{\mu_{rd,a}}| \geq S - (a+1)t - (a-1)b$. Furthermore, since at most b servers are malicious, $\Sigma_{\mu_{rd,a}} \cap \Sigma_{rd'}$ contains at least $S - (a+1)t - ab$ servers.

Moreover, since $a \in [1, R+1]$ and $R < \frac{S+b}{t+b} - 2$, we have $S - (a+1)t - ab \geq 1$. □

LEMMA 6.12. *Assume that:*

1. *$maxTS_{rd}$ is admissible with degree $a \in [1, R+1]$ in some non-malicious* read *rd,*

2. a complete non-malicious read *rd' by reader r_j follows rd,*

3. there is a set $X \subseteq \Sigma_{\mu_{rd,a}}$ of at least $t+1$ non-malicious servers, such that for all $s_i \in X$, s_i sends message $m_i \in \mu_{rd,a}$ with $r_j \in m_i.updated$.

Then, rd' does not return a value smaller than $maxTS_{rd}$.

LEMMA 6.13. ***("read-read" atomicity (SWA4).*** *If non-malicious* reads *rd_1 and rd_2 return timestamps ret_{rd_1} and ret_{rd_2}, respectively, and if rd_2 follows rd_1, then $ret_{rd_2} \geq ret_{rd_1}$.*

THEOREM 6.14. *The algorithm of Figure 6.1 is a fast implementation of an atomic SWMR register in the arbitrary failure model.*

Proof. Atomicity follows from Lemmas 6.5, 6.6, 6.9 and 6.13. Moreover, it is obvious that our implementation satisfies Termination — conditions in lines 6 and 13 are non-blocking since we assume at least S-t correct (and non-malicious) servers. Finally, our implementation is fast: all operations involve a single communication round-trip between a client and servers. □

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

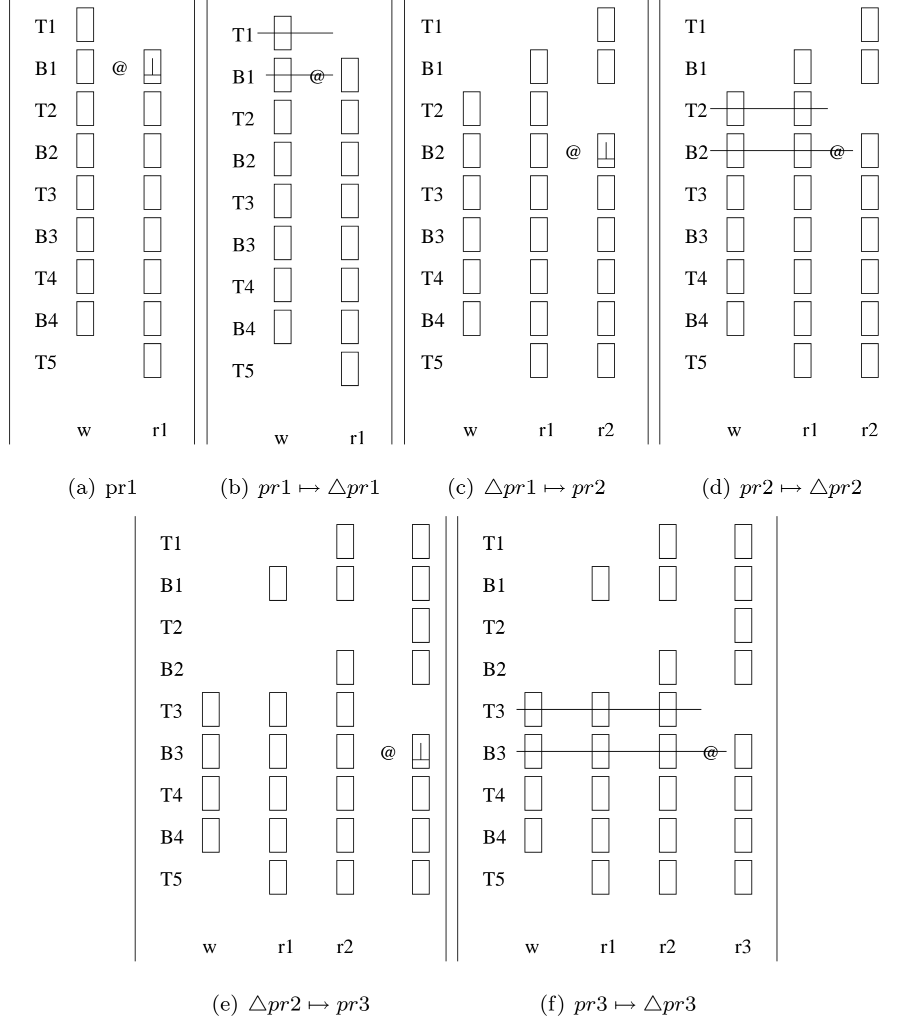
PROPOSITION 6.15. *Let $t \geq 1$, $b \geq 0$ and $R \geq 2$. If $(R+2)t+(R+1)b \geq S$, then there is no fast atomic register implementation.*

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 if I makes use of digital signatures). 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.

Proof. Given that $(R+2)t+(R+1)b \geq S$, we can partition the set of servers into $2R+3$ subsets, which we call *blocks*, denoted by T_i $(1 \leq i \leq R+2)$ and B_j $(1 \leq j \leq R+1)$, such that each of the blocks T_i (resp., B_j) is of size less than or equal to t (resp., b). We illustrate a particular instance of the proof in Figure 6.2 and Figure 6.3, where $R=3$ and the set of servers are partitioned into nine blocks, T_1 to T_5 and B_1 to B_4. In these figures, we denote the arbitrary failure of B_i by @.

Partial writes. Consider a partial run wr in which w completes write(1). The operation skips T_{R+2}. We define a series of partial executions 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 operation write(1) 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 the write completes, and hence, (3) wr and wr_1 differ only at w.

Appending reads. Partial run pr_1 extends wr by having block B_1 failing upon completion of write(1) and appending a complete read by r_1 that skips block T_1 (Fig. 6.2(a)). B_1 fails in such a way that it behaves as if it never received any write message (i.e., a message from operation 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 (a) wr and wr_2 differ at w and at blocks T_1 and B_1, (b) r_1 does not receive any message from writer



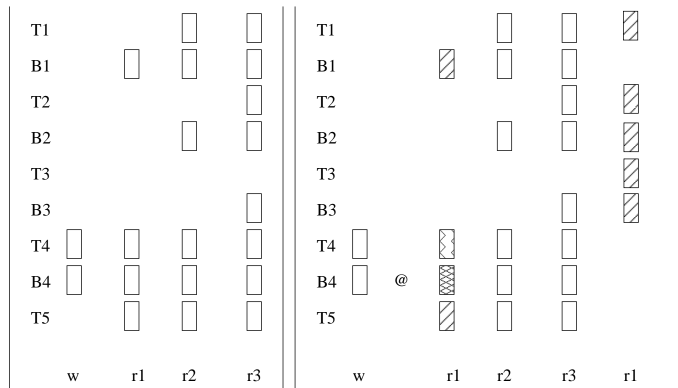
(a) pr1 (b) $pr1 \mapsto \triangle pr1$ (c) $\triangle pr1 \mapsto pr2$ (d) $pr2 \mapsto \triangle pr2$

(e) $\triangle pr2 \mapsto pr3$ (f) $pr3 \mapsto \triangle pr3$

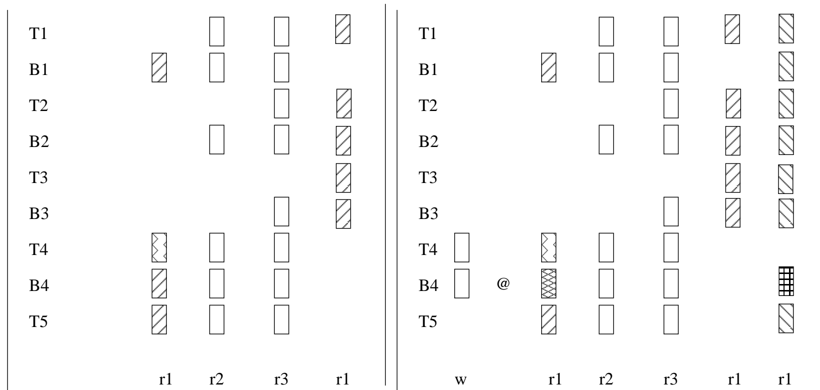
FIG. 6.2. *Partial runs* pr_i *and* $\triangle pr_i$

w and block T_1 in both executions and (c) r_1 received the same message from block B_1 in both executions. By wait-freedom property and since w can fail by crashing, r_1's read in $\triangle pr_1$ must return some value x, since it cannot wait for the completion of the writer's invocation, nor a message from w. Since r_1 cannot distinguish $\triangle pr_1$ from pr_1, r_1 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.

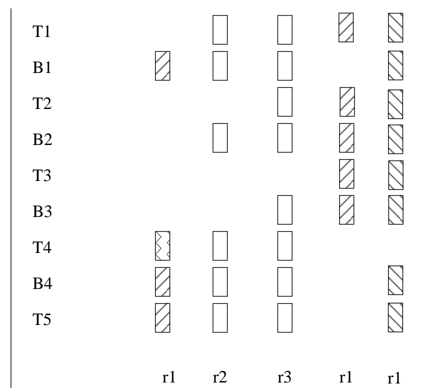
Starting from $\triangle pr_1$, we iteratively define the following partial executions for $2 \leq i \leq R$. Partial run pr_i extends $\triangle pr_{i-1}$ by: (1) block B_i failing in such a way that it behaves as if it never received any message (loses memory) and (2) appending a complete read by r_i 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 executions r_i receives the same messages from B_i. More precisely, partial run $\triangle pr_i$



(a) $\triangle pr3$ (b) pr^A



(c) pr^B (d) pr^C



(e) pr^D

non–malicious block that replies to the first read of r1, but r1 does not receive replies

malicious block that replies to the first read of r1 and r1 receives the replies

non–malicious block that replies to the first read of r1 and r1 receives the replies

malicious block that replies to the second read of r1 and r1 receives the replies

non–malicious block that replies to the second read of r1 and r1 receives the replies

(f) Legend

Fig. 6.3. *Partial executions:* pr^A, pr^B, pr^C *and* pr^D

extends wr_{i+1} by appending the following i reads one after the other: 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. Here, the first $i-1$ appended reads are incomplete whereas the last one is complete. Figure 6.2 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 wait-freedom, in $\triangle pr_2$ r_2 must return some value, say x_2. However, since r_2 cannot distinguish pr_2 from $\triangle pr_2$, r_2 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 by 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 object is faulty.

Partial run pr^A. Consider again partial run $\triangle pr_R$, i.e., partial run 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 $\{T_j | h \leq j \leq R-1\} \cup \{B_j | h+1 \leq j \leq R-1\}$, whereas 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 servers in T_{R+1} and B_{R+1} receive the WRITE message from write(1). Consider the following partial run pr^A which differs from $\triangle pr_R$ in the following:

1. Upon reception of message from write(1) operation, 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. Moreover, after completion of read by r_R,
2. r_1 receives the READACK messages from T_{R+2} and B_1 (that were in transit in $\triangle pr_R$) and B_{R+1} (i.e., from the Byzantine faulty objects),
3. 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, and
4. 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}, i.e., from the only benign servers whose state was modified by write(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 6.3). 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 executions 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 executions. By atomicity, r_1's read returns (the initial storage value) $\perp$ 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 executions 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 executions: (1) pr^C is constructed by

extending pr^A with another complete read by r_1, which skips T_{R+1} (as in pr^A, in pr^C B_{R+1} always replies to r_1 as if it never received any WRITE message), and (2) pr^D is constructed by extending pr^B with another complete read by r_1, which skips T_{R+1} (Figure 6.3). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips T_{R+1} (i.e., 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 follows r_R's read which returns 1. Clearly, partial run pr^C violates atomicity. □

7. Multiple Writers. In the impossibility proof below, we use two simple properties of MWMR atomic register which can be easily deduced from atomicity (see Section 3.1):

- (Property **P1**) in any partial run, 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**) in any partial run, 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 (in the following, W denotes the number of writers). The proof is written for the crash-stop model. But by extension the impossibility directly applies to the arbitrary failure model.

PROPOSITION 7.1. *Let $t \geq 1$, $R \geq 2$ and $W \geq 2$. 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 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 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 operations: (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 run^{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 $r2$ that skips s_{i1}. From property P2, it follows that $r2$ returns 1. Similarly we construct a partial run run'' which extends run^{i1+1} with a read by $r2$ that skips s_{i1}. Recall that, only $w1$, $w2$, $r1$ and s_{i1} can distinguish run^{i1} from run^{i1+1}. Since $r2$ skips s_{i1} in both run' and run'', $r2$ cannot distinguish the two partial runs. Thus $r2$ returns 1 in run''. However, $r1$ 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 operation at a time. Thus we cannot construct partial runs with concurrent writes in the single-writer case.

8. Related Work. A seminal SWMR crash-tolerant atomic register implementation assuming a majority of correct processes, known as ABD, was presented by Attiya, Bar-Noy and Dolev in [4]. In ABD, all write operations are fast; however, read operations always take two communication round trips between a client and servers. In this paper, we show that having fast read operation in a SWMR atomic implementation is possible, yet it comes with a somewhat steep price — a limited number of readers.

In [20, 10], ABD was extended to quorum system-based implementations of MWMR atomic register. In both these MWMR implementations, a read or write operation requires at least two rounds trips. In [23], Lynch and Shvartsman implement a MWMR register in a dynamic system, where processes can join or leave the set of servers implementing the register. However, even in executions where no process joins or leaves the set of server, a read or write operation in [23] requires at least two round-trips. Thus, the time-complexities of these implementations are consistent with our result on the impossibility of fast MWMR implementations when servers may fail.

Our results adapt the classical theorem "atomic reads must write", stated in a shared-memory context [18, 5], to a message-passing context. In particular, 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 [5]. A similar result appears in the context of atomic register implementations over weaker *regular* ones

[18]. Namely, in such atomic register implementations, 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', which is possible when reading from regular registers.

Assuming a message-passing system, Fan and Lynch show [11] that 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, even in one round-trip, a read can modify at least $S - t > t$ servers (assuming a majority of correct servers).

There is a prolific line of research in Byzantine fault tolerant atomic register implementations in message-passing systems, e.g., [24, 25, 6, 14, 3, 15], with a typical focus on providing optimal resilience (in our model, this amounts to $S = 2t + b + 1$ servers [25]). The work of Malkhi and Reiter [24] casts the ABD algorithm to the MWMR Byzantine context, featuring both two round-trip writes and two round-trip reads, using writer's digital signatures, which we also use in the Byzantine-tolerant version of our implementation. A MWMR implementation by Martin et al. [25] introduces the "Listeners" pattern in which readers, roughly speaking, subscribe to updates from servers. Like in [25], in our implementation readers modify the servers' states, but receive no updates since this would, intuitively, violate the requirement for a fast implementation. Cachin and Tessaro propose in [6] a Byzantine-tolerant variant of Rabin's information dispersal algorithm [27] to minimize the storage blowup inherent to data replication. To this end, a MWMR implementation of [6] relies on communication among servers which, in a sense, prohibits fast operations. SWMR implementations that allow fast "best-case" read/write operations, i.e., operations that execute in synchronous periods, with few failures and no read/write concurrency, were presented in [14, 15]. In contrast, in this paper we consider the problem of allowing *all* operations to be fast while assuming the general, unrestricted asynchronous system. Not surprisingly, our limitations related to the number of readers are incurred by worst-case interleaving among different, concurrent operations, with roots in asynchrony and (possible) failures. Note that [14, 15] as well as the MWMR implementation of Aiyer et al. [3], renounce digital signatures. In this light, it is important to note that the existence of fast Byzantine-resilient atomic register implementations that do not rely on digital signatures remains an open problem.

After the appearance of the preliminary version of this work [8], several papers extended the notion of fast implementations. For SWMR implementations where the servers can only fail by crashing, Georgiu et al. propose in [13] how to circumvent our fast implementation lower bound ($R < \frac{S}{t} - 2$) by permitting some reads that are not fast (called *slow* reads). More specifically, in the *semi-fast* implementation of [13], the readers are grouped into virtual nodes where readers in the same node possess the same virtual identifiers. Then, as long as there are at most $\frac{S}{t} - 2$ virtual identifiers in the system (irrespective of the number of readers), most read operations are fast (and there is at most one slow read operation per write operation). In [12], the same authors investigate quorum system-based fast and semi-fast implementations. The paper shows that for *robust* quorum systems (i.e., quorum systems that remains available when one of the servers fails) and in presence of arbitrary number of readers, it is impossible to implement fast or semi-fast SWMR registers. The paper then presents a *weak-semifast* implementation that allows multiple slow read operations per write operation. In a recent work [9], Englert et al. investigate the possibility of MWMR implementations where most operations are fast, by assigning some additional

responsibilities at server for ordering operations.

Thus, our results in this paper have initiated a line of work that investigates the trade-off between the efficiency of atomic register implementations and the bounds on the number of readers and writers. This is not surprising since: a) atomic read/write registers are seen as a fundamental abstraction in building practical distributed storage and file systems (see e.g., [30, 29]) and b) our results demonstrate a fundamental limitation on the number of readers that an asynchronous fast SWMR atomic implementations can support, as well as the impossibility of fast MWMR atomic implementations. In a sense, the line of research that stems from our work seeks for practically applicable atomic register implementations by circumventing our results, while possibly allowing for some operations to be fast.

9. Summary. This paper establishes the exact conditions required for a fast implementation (an implementation in which all operations complete in a single round-trip) 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.

Finally, it would be interesting to look into optimal register implementations with respect to other complexity metrics (e.g., message complexity). This is left as future work.

REFERENCES

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 59–72, 2005.

[2] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[3] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 7–19, 2007.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[6] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine dis-

tributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124, 2006.

[7] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 45–58, 2006.

[8] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 236–245, 2004.

[9] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 240–254, 2009.

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

[11] R. Fan and N. Lynch. Efficient replication of large data objects. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 75–91, 2003.

[12] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *Proceedings of the 22nd International symposium on Distributed Computing*, pages 289–304, 2008.

[13] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.

[14] Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, 2006.

[15] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

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

[17] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[18] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, 1986.

[19] Leslie Lamport. Lower bounds for asynchronous consensus. *Future Directions in Distributed Computing*, pages 22–23, 2003.

[20] Nancy Lynch and Alexander Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

[21] Nancy A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[22] Nancy A. Lynch and Mark R.Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[23] Nancy A. Lynch and Alexander A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 173–190, 2002.

[24] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 51–58, 1998.

[25] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, October 2002.

[26] M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[27] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[28] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[29] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[30] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.