How Fast Can a Very Robust Read Be?

[Extended Abstract]

Rachid Guerraoui
IC, EPFL
CH-1015 Lausanne, Switzerland, and
CSAIL, MIT
Cambridge, MA 02139, USA
rachid.guerraoui@epfl.ch

Marko Vukolić
IC, EPFL
CH-1015 Lausanne, Switzerland
marko.vukolic@epfl.ch

ABSTRACT

This paper studies the time complexity of reading unauthenticated data from a distributed storage made of a set of failure-prone base objects. More specifically, we consider the abstraction of a robust read/write storage that provides wait-free access to unauthenticated data over a set of base storage objects with t possible failures, out of which at most b are arbitrary and the rest are simple crash failures.

We prove a 2 communication round-trip lower bound for reading from a *safe* storage that uses at most $2t + 2b$ base objects, independently of the number or round-trips needed by the writer. We then prove the lower bound tight by exhibiting a *regular* storage that uses $2t + b + 1$ base objects (optimal resilience) and features 2 communication round-trips for both read and write operations.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*

General Terms

Algorithms, Performance, Reliability, Theory

Keywords

Storage emulations, Arbitrary failures, Optimal resilience, Time-complexity

PODC'06, July 22-26, 2006, Denver, Colorado, USA.

1. INTRODUCTION

We study *robust* storage implementations that provide wait-free [11] access to unauthenticated data in an asynchronous environment over a set of S base storage objects: t of these might fail, out of which b ($b > 0$) might be arbitrary [12] (also called Byzantine [14]) failures. The storage we consider implements the celebrated single-writer multi-reader (SWMR) *register* abstraction in a hostile environment [13].

Motivated by the availability of networks of commodity disks, such implementations have been widely studied in the last decade. Whereas original storage implementations tolerated only crash failures of the base objects (i.e., the case where $b = 0$) [3], more recent implementations tolerated arbitrary object failures ($b \neq 0$) [1, 4–9, 12, 18]. The *optimal resilience* for such implementations was shown to be $S = 2t + b + 1$, assuming $b = t$ [18], but can easily be extended to the general $b \neq t$ case.

Maybe surprisingly given the large body of literature in the area, there is no general result on the time complexity of *reading* in such a storage. This is particularly surprising since the *read* operation is considered the most frequent in practice. Typically, we would like to determine the latency of the read operation, which can be measured as the number of *communication round-trips* (or simply *rounds*) needed in the worst case between the reader and the base objects, before a value can be returned.

The complexity of *writing* has actually been carefully studied. The tight lower bound on the worst-case complexity of the write operation was shown to be 2 rounds when at most $2t + 2b$ of these objects are used; if more than $2t + 2b$ base objects are available, then a single round suffices [1][1]. This lower bound is general, since it was established for any *safe* storage; a storage that simply ensures that a read returns the last written value if it is not concurrent with any write [13]. In fact, it was also shown in [1] that this bound is tight, even for a stronger *regular* storage: one that is not only safe but also only returns written values, even if there is a concurrent writer [13].

There is no such general picture for a *read* operation. In fact, the complexity of reading was studied, but only in some specific cases. For instance, it was shown that, for any safe storage, when readers do not modify the state of the base

[1] [1] also assumes $b = t$, but, again, it is not difficult to extend its results to the general $b \neq t$ case.

objects, the optimal read complexity with less than $2t + 2b$ base objects is $b + 1$ rounds [1]. It was also shown that the optimal complexity of an *atomic* read, providing the illusion of instantaneous access [13], is one round when (a) more than $R(t + b) + 2t + b$ base objects are available (where R is the number of readers) and if the write also takes one round [7], or (b) the read does not encounter any contention, asynchrony or many failures [8, 9].

But what is the general complexity of a read operation? The contribution of this paper is to address this fundamental question for the worst case, and in particular for an optimally resilient storage.

- We prove a 2-round lower bound for reading from a *safe* storage that uses at most $2t + 2b$ base objects, independently of the number or rounds needed by the writer.
- We then prove the lower bound tight even for a *regular* storage that is optimally resilient and uses $2t + b + 1$ base objects.

We address this question first by considering a *data centric* storage [17] in which the base objects represent (active) disks (i.e., atomic read-modify-write objects) that do not communicate among each other, nor initiate unsolicited messages to the clients (i.e., push messages). Later in the paper, we show how to migrate our lower bound to a *server centric* model where the data is stored within first class processes that can send unsolicited messages (Section 6). The model we describe in Section 2 allows us to easily extend our data centric model into a server centric one.

We proceed through three major steps.

1. We first prove in Section 3 that $S = 2t + 2b$ base objects are insufficient for a safe SWSR wait-free storage implementation in which every read takes one round-trip (we say it is *fast*). Roughly, our proof derives a contradiction from three runs that are indistinguishable to the reader. In the first run, a read is concurrent with the write and all base objects are correct; in the second one, the write precedes the read but malicious base objects forge their state to simulate the concurrency of the first run; finally, in the third run, malicious base objects forge their state to simulate the above mentioned concurrency, although the write is never invoked. As the read must return the same value in all three runs, without invoking additional communication round-trips, safety is violated in either the second, or the third run. In our proof, we do not make any assumption on the time-complexity of the write operation. We assume however that data is not authenticated [20]; data authentication is considered a source of overhead [15, 19], and we would typically like to avoid using authentication when seeking optimal time-complexity. If we permit data authentication, then regular storage can be implemented fairly simply, while achieving both optimal resilience and fast reads/writes [16].

2. We then describe in Section 4 an optimally resilient SWMR safe storage algorithm that features optimal (worst-case) time complexity for both read and write operations: 2 rounds. This algorithm is interesting in its own right as it contradicts the conjecture of [1] suggesting that $b + 1$ rounds are needed in order to read from a safe storage. The algorithm uses novel techniques to combine optimal resilience with optimal time-complexity. Roughly, unlike in traditional safe storages we know of, in both of their communication rounds, readers both change the state of the base objects and read their current state. The writer does the same in its first round, along with simply writing in the second round. Basically, by allowing readers to change the state of the base objects, twice in a row, we allow the readers to carefully filter the responses from malicious base objects that may be trying to mislead the reader.

3. Finally, we show in Section 5 how to modify our safe implementation and obtain a regular one without sacrificing neither optimal resilience nor optimal time-complexity. Our regular implementation relies however on the fact that base objects keep all the values they receive from the writer (which is not the case with our safe implementation). Although some very practical storage systems rely on the same assumption [8], this might raise issues of storage exhaustion and needs careful garbage collection. Comparable data-centric regular wait-free storage implementations that do not rely on this assumption are either not optimally resilient [2], or do not feature the optimal (worst-case) time-complexity of the read operation [9].

For the lack of space, correctness proofs of our algorithms are given in the full version of the paper [10].

2. MODEL

The distributed system we consider consists of three *disjoint* sets of processes: a set *objects* of size S containing processes $\{s_1, ..., s_S\}$ and representing the base storage elements; a singleton *writer* containing a single process $\{w\}$; and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. The set *clients* is the union of the sets *writer* and *readers*. We assume that every client may communicate with any process by message passing using point-to-point reliable communication channels. However, objects cannot communicate among each other, nor send messages to clients other than in reply to clients' messages. (We will come back to this assumption later in the paper in Section 6.)

For presentation simplicity, we also assume a global clock, which, however, is not accessible to either clients or objects, for these have an asynchronous perception of their environment.

2.1 Runs and Algorithms

The state of the communication channel between processes p and q is viewed as a set $mset_{p,q} = mset_{q,p}$ 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 deterministic automata, where A_p is the automata assigned to process p. Computation of *non-malicious* processes 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 $sp =< p, M >$, process p atomically does the following (we say that p *takes* step sp): (1) removes the messages in M from $mset_{p,*}$, (2) applies 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 $mset_{p,*}$. Moreover, a non-malicious object s_i may put the output messages in $mset_{s_i,c}$ in step $sp =< p, M >$ only if s_i received in sp a message $m \in M$ sent by the client c (we relax this restriction in Section 6, when discussing the server centric model). A *malicious* process p can perform arbitrary *actions*: (1) it can remove/put arbitrary messages from/into $mset_{p,*}$ and (2) it can change its state in an arbitrary manner.

Given any algorithm A, a *run* of A is an infinite sequence of steps of A taken by non-malicious processes, and actions of malicious processes, such that the following properties hold for each non-malicious process p: (1) initially, for each non-malicious process q, $mset_{p,q} = \emptyset$, (2) the current state in the first step of p is a special state *Init*, (3) for each step $< p, M >$ of A, and for every message $m \in M$, p is the receiver of m and $\exists q, mset_{p,q}$ that contains m immediately before the step $< p, M >$ is taken, and (4) if there is a step that puts a message m in $mset_{p,*}$ such that p is the receiver of m and p takes an infinite number of steps, then there is a subsequent step $< p, M >$ such that $m \in M$. A *partial run* is a finite prefix of some run. A (partial) run 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*.

We say that a *non-malicious* process p is *correct* in a run r if p takes an infinite number of steps of A in r. Otherwise a *non-malicious* process p is *crash-faulty*. We say that a *crash-faulty* process p *crashes* at step sp in a run, if sp is the last step of p in that run. *Malicious* and *crash-faulty* processes are called *faulty*. In any run of our model, at most t objects might be faulty. At most b out of these t objects may be *malicious*. In this paper we assume $b > 0$. An algorithm that assumes a total number of objects S equal to $2t + b + 1$ is said to be *optimally resilient*.

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

2.2 Robust Storage

A storage abstraction is a READ/WRITE data structure. It provides two operations: WRITE(v), which stores v in the storage, and READ(), which returns the value from the storage. We assume that each client invokes at most one operation at a time (i.e., does not invoke the next operation until it receives the response for the current operation). Only readers invoke READ operations and only the writer invokes WRITE operations. We further assume that the initial value of a storage is a special value $\perp$, which is not a valid input value for a WRITE operation. We say that an operation op is *complete* in a (partial) run if the run contains a response step for op. In any run, we say that a complete operation $op1$ *precedes* operation $op2$ (or $op2$ *succeeds* $op1$) if the response step of $op1$ precedes the invocation step of $op2$ in that run. If neither $op1$ nor $op2$ precedes the other, the operations are said to be *concurrent*.

An algorithm *implements* a robust *safe* (resp., *regular*) storage if every run of the algorithm satisfies *wait-freedom* and *safety* (resp., *regularity*) properties. Wait-freedom states that if a client invokes an operation and does not crash, eventually the client receives a response (i.e., operation completes), independently of the possible crashes of any other client. Here we give a definition of safety and regularity for the SWMR storage.

In the single-writer setting, the WRITE operations in a run have a natural ordering which corresponds to their physical order. Denote by wr_k the k^{th} WRITE in a run ($k \geq 1$), and by val_k the value written by wr_k. Let $val_0 = \perp$.

We say that a partial run satisfies *safety* if every READ operation rd that is not concurrent with any WRITE operation returns val_k such that wr_k precedes rd and for no $l > k$ wr_l precedes rd, or val_0 in case there is no such a value; a READ concurrent with a WRITE is allowed to return any value.

Similarly, we say that a partial run satisfies *regularity* if the following properties hold: (1) if a READ returns x then there is k such that $val_k = x$, (2) if a READ rd is complete and it succeeds some WRITE wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$, and (3) if a READ rd returns val_k ($k \geq 1$), then wr_k either precedes rd or is concurrent with rd.

In the following, under the notion of *implementation*, we assume, by default, a wait-free storage that stores unauthenticated data in an asynchronous communication model.

2.3 Fast READ

Basically, we say that a READ operation is *fast* if it completes in a single communication round-trip. In every communication round-trip (we simply say round) rnd of an operation op invoked by the client c:

1. The client c sends messages to all objects. This is indeed without loss of generality because we can simply model the fact that messages are not sent to certain objects by having these objects not change their state or reply.

2. Objects, on receiving such a message, reply to the reader (resp., writer) before receiving any other messages (as dictated by our model).

3. When the invoking client receives a sufficient number of such replies, the round (rnd) terminates.

Note that, since any number of clients can crash, we can construct partial runs in which no client receives any message from any other client. In our proof in Section 3 we focus, without loss of generality, on such partial runs. Moreover, since up to t objects might crash in our model, ideally, in every round rnd the invoking client can only wait for reply messages from $S - t$ correct objects.

A READ rd is *fast* if rd completes in the step in which the first round of rd terminates. We say that a storage implementation I is a *fast READ* implementation, if every complete READ operation in every run of I is fast. For a fast READ implementation, we can say without ambiguity that the messages sent by a reader, on invoking a READ, are of type READ, and the messages sent by a process to the reader, on receiving a READ message, of type READACK.

3. LOWER BOUND

We prove in this section that there is no safe storage implementation with at most $2t + 2b$ objects in which every READ is *fast*. In our proof, we assume that a set of readers is a singleton.

Proposition 1. There is no fast READ implementation I of a single reader (SWSR) safe storage that makes use of less than $2t + 2b + 1$ objects.

Preliminaries. Recall first that w denotes the writer, r_1 the reader, and s_i for $1 \leq i \leq S$ denote the objects. Suppose, by contradiction, that there is a safe storage implementation I that uses at most $2t + 2b$ objects, such that, in every (partial) run of I every READ operation completes in a single round (i.e., every READ is *fast*).

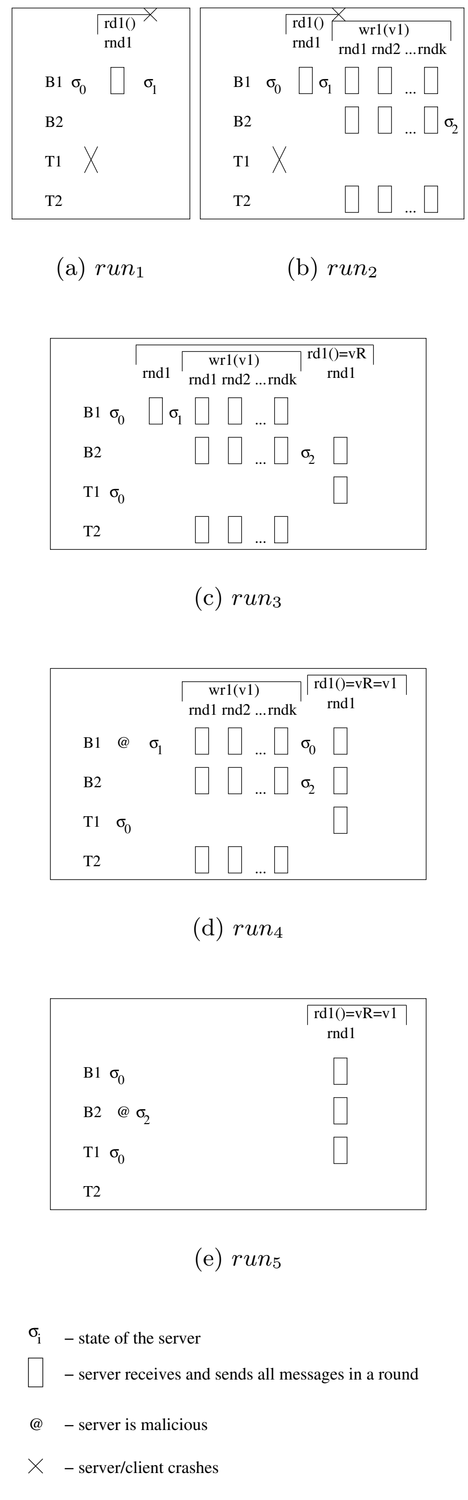
We partition the set of objects into four distinct subsets (which we call *blocks*), denoted by T_1 and T_2, each of size exactly t, and B_1 and B_2 of size at least 1 and at most b. Note that we assume $S \geq 2t + 2$, without loss of generality since the number of objects for any implementation I must conform with the optimal resilience lower bound of $S \geq 2t + b + 1$ [18] (recall that we assume $b > 0$). Therefore, without loss of generality, we can assume that each of the blocks T_1, T_2, B_1 and B_2 contains at least one object. We refer to the initial state of every correct object as σ_0.

We say that a message m of a round rnd of an incomplete operation *op skips* a set of blocks BS in a partial run (where $BS \subseteq \{T_1, T_2, B_1, B_2\}$), if (1) no object in any block $BL \in BS$ receives m in round rnd of *op* in that partial run, (2) all other objects receive m in round rnd of *op* and reply to that message, and (3) *all these reply messages are in transit*. We say that a *complete operation op skips* a set of blocks BS in a partial run, if (1) no object in any block $BL \in BS$ receives any message in any round of *op* in that partial run, (2) all objects that are not in any block $BL \in BS$ receive the message from the invoking client in every round of *op* and reply to such message, and (3) the invoking client *receives all these reply messages* and, finally, returns from the invocation.

Block diagrams. We illustrate the idea behind the proof in Figure 1. We depict a round rnd of an operation *op* through a set of rectangles, arranged in a single column. In the column corresponding to some round rnd of an operation *op*, we draw a rectangle in the particular row, if all objects in the corresponding block BL have received the message from the client in round rnd of *op* and have sent reply messages, i.e., we draw a rectangle in the row corresponding to BL if round rnd of *op* does not skip BL.

Proof. To exhibit a contradiction, we construct a partial run of the safe implementation I that violates safety. More specifically, we exhibit a partial run in which some READ returns a value that was never written.

- Let run_1 be the partial run in which all objects are correct except T_1 that crashes at the beginning of run_1. Furthermore, let rd_1 be the READ operation by the reader (r_1) and no other operation is invoked in run_1. In run_1, r_1 crashes and rd_1 skips B_2, T_1 and T_2. After B_1 sends *readack* to r_1, run_1 ends. We refer to the state of object B_1, at the end of run_1 as to σ_1.



(a) run_1 (b) run_2

(c) run_3

(d) run_4

(e) run_5

(f) Legend

Figure 1: Illustration of the runs used in the proof of Proposition 1

- Let run_2 extend run_1 by appending a WRITE wr_1 invoked by the correct writer to write a value $v_1 \neq \perp$ in the storage. By our assumption on I (I is wait-free), wr_1 completes in run_2, say at time t_1 after invoking a finite number (k) of rounds. Therefore, wr_1 skips T_1, and completes (at latest) after the writer receives the replies in round k from correct objects (B_1, B_2, and T_2). We refer to the state of the correct object B_2 at time t_1 as to σ_2.

- Let run_2' be the partial run that ends at t_1, such that run_2' is identical to run_2 up to time t_1, except that in run_2' object T_1 does not crash, but, due to asynchrony, all messages sent by the writer to T_1 during wr_1 remain in transit. Since the writer cannot distinguish run_2 from run_2', wr_1 skips T_1 and completes in run_2' at t_1.

- Let run_2'' be the partial run identical to run_2' up to time t_1, except that, in run_2'', (1) the reader does not crash in run_2'', but, due to asynchrony, all messages that were in transit in run_2' are delayed in run_2'' until after t_1, and (2) object T_2 crashes at t_1. By our assumption on the wait-freedom of I, rd_1 completes in run_2'' at t_2 after receiving *readack* messages from correct objects (B_1, B_2 and T_1) and returns some value v_R, skipping T_2.

- Let run_3 be the partial run identical to run_2'', except that, in run_3, T_2 does not crash, but, due to asynchrony, all messages exchanged between r_1 and T_2 during rd_1 are delayed until after t_2. Since r_1 cannot distinguish run_3 from run_2'', rd_1 completes in run_3 at t_2 and returns v_R. Note that in run_3 all objects are correct.

- Let run_4 be the partial run similar to run_3, except that, in run_4: (1) rd_1 is invoked only after wr_1 completes (after t_1) (2) B_1 is malicious and forges it state to σ_1 at the beginning of the run (as if it received a round 1 message of rd_1 from the reader, as in run_3), before wr_1 is invoked, (3) after t_1, a READ rd_1 is invoked and (4) at t_1, B_1, before replying to rd_1, forges its state to σ_0, the initial state of correct objects. Other messages are delivered as in run_3, in particular, messages exchanged between r_1 and T_1 are transit in run_4. Note that wr_1 cannot distinguish run_4 from run_3 and hence, wr_1 completes in run_4 at t_1. Note also that, rd_1 is invoked after wr_1 completes, so safety implies that rd_1 must return v_1. However, note that in run_3 and run_4 the reader receives in rd_1 the identical messages and, since the processes do not have access to global clock, r_1 (as well as the correct objects B_2, T_1 and T_2) cannot distinguish run_4 from run_3. Therefore, in run_3 and run_4 rd_1 returns the same value, i.e., v_R, that, by safety, must equal v_1.

- Finally, consider the partial run run_5 in which wr_1 is never invoked, but B_2 is malicious and forges its state to σ_2 at the beginning of the run. READ rd_1 is invoked in run_5 as in run_4. Since, upon receiving *readack* messages from B_1, B_2 and T_1, the reader receives identical information as in run_4, the reader cannot distinguish run_4 from run_5 (neither can correct objects B_1, T_1 and T_2), and rd_1 completes in run_5 and returns a $v_R = v_1$. However, by safety, in run_5, rd_1 must return $\perp$. Since $v_1 \neq \perp$, safety is violated in run_5.

4. SAFE IMPLEMENTATION

Our algorithm uses $S = 2t + b + 1$ objects (optimal resilience) to implement a SWMR safe storage. Besides its optimal resilience, our implementation features optimal (worst-case) time complexity for both READ and WRITE operations, i.e., two communication round-trips. In fact, the existence of our algorithm proves the following proposition:

Proposition 2. There is an optimally resilient implementation I of a SWMR safe storage such that, in every partial run of I, every (READ/WRITE) operation completes in at most two communication round-trips.

In the following, we give a detailed description of our algorithm. Due to lack of space, we omit the correctness proof (see the full paper [10]).

4.1 Overview

Both the READ and the WRITE operations take at most two rounds. In each round, the client (reader or writer) sends a message to all objects. A round terminates at the latest when the client receives the responses from $S - t$ correct objects. In the first round, the writer, in addition to writing data, reads control data from the objects. Readers write control data and read data written by the writer in both rounds.

The base objects maintain the following variables (we call *fields*) pw, w and the array $tsr[1, ..., R]$ (where R is the number of readers). In the pw field, objects store a timestamp-value pair $tsval$ of the form $\langle ts, v \rangle$. In the w field, objects store the following pair: $\langle tsval, tsrarray[1..S] \rangle$. Fields pw and w are written by the writer, and each field $tsr[j]$ by the reader r_j.

In both rounds of the READ, the reader r_j: (1) increases its local timestamp tsr_j' and stores it in the objects' $tsr[j]$ field and (2) reads the objects' fields pw and w.

In the first round of the WRITE (called PW), the writer, writing the value v: (1) increases its timestamp ts, (2) assigns the timestamp-value pair $\langle ts, v \rangle$ to its variable pw', (3) writes pw' to the objects' pw fields and the last copy of w' to the objects' w fields, (4) reads the values of objects' fields $tsr[*]$ that are written by readers and (5) adds the values $tsr[*]$ to the array (of arrays) $currenttsrarray$. Upon receiving $S - t$ responses from different objects in round PW, the writer proceeds to the second round, W.

In the second round of the WRITE, the writer: (1) assigns $w' := \langle pw', currenttsrarray \rangle$ and (2) writes pw' to the objects' pw fields and w' to the objects' w fields. Upon receiving $S - t$ responses from different objects in round W, the WRITE completes. The objects change the values of $tsr[*]$, pw, and w only if these are newer than the copies already stored (Figure 3).

The WRITE implementation is given in Figure 2. In the following, we detail the READ implementation, since it is slightly more involved and the main focus of this paper.

4.2 READ implementation

The full READ implementation is given in Figure 4. In the following, unless explicitly stated otherwise, we refer to Figure 4.

Initialization:
1: $inittsrarray[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
2: $pw := \langle 0, \perp \rangle$; $ts := 0$; $w := \langle pw, inittsrarray \rangle$

WRITE(v) **is** {
3: **inc**(ts); $currenttsrarray := inittsrarray$
4: $pw := \langle ts, v \rangle$
5: send $PW\langle ts, pw, w \rangle$ to all objects
6: **wait for** $PW\_ACK_i\langle ts, tsr \rangle$ from $S - t$ different objects
7: $w := \langle pw, currenttsrarray \rangle$
8: send $W\langle ts, pw, w \rangle$ message to all objects
9: **wait for** $WRITE\_ACK_i\langle ts \rangle$ from $S - t$ different objects
10: **return**(OK)

upon reception of $PW\_ACK_i\langle ts, tsr \rangle$ from s_i
11: $currenttsrarray[i] := tsr$
}

Figure 2: SWMR safe storage: WRITE implementation - code of the writer

Initialization:
1: $ts := 0$; $inittsrarray[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
2: $pw := \langle 0, \perp \rangle$; $w := \langle pw, inittsrarray \rangle$; $tsr[j] := 0$, $1 \leq j \leq R$

3: **upon** reception of $PW\langle ts', pw', w' \rangle$ message from the writer **do**
4: **if** $ts' > ts$ **then**
5: $ts := ts'$; $pw := pw'$; $w := w'$
6: send $PW\_ACK_i\langle ts, tsr \rangle$ to the writer
7: **endif**

8: **upon** reception of $W\langle ts', pw', w' \rangle$ message from the writer **do**
9: **if** $ts' \geq ts$ **then**
10: $ts := ts'$; $pw := pw'$; $w := w'$
11: send $WRITE\_ACK_i\langle ts \rangle$ to the writer
12: **endif**

13: **upon** reception of $READk\langle tsr' \rangle$ mess. from r_j ($k \in \{1, 2\}$) **do**
14: **if** $tsr' > tsr[j]$ **then**
15: $tsr[j] := tsr'$
16: send $READk\_ACK_i\langle tsr[j], pw, w \rangle$ to the reader r_j
17: **endif**

Figure 3: SWMR safe storage: code of object s_i

As we previously mentioned, in both rounds of the READ, the reader: (1) increases its local timestamp tsr'_j (lines 9 and 12), and stores it in the objects' $tsr[j]$ fields using READ1 (in the first round), or READ2 (in the second round) messages (lines 10 and 13) and (2) reads the objects' fields pw and w by receiving $READ1\_ACK_*$, or $READ2\_ACK_*$ messages (lines 11, 14 and 21-26).

When the reader receives a timestamp-value pair pw' from the pw field of object s_i (we say s_i *reports* pw'), the reader adds i, the index of object s_i, to the set $RPW(pw')$ that is initially empty. Similarly, if s_i reports a tuple w' in its w field, the reader adds i to the set $RW(w')$. If this occurs in the first round of the READ, the reader also adds i to $FirstRW(w')$. (lines 22, 23 and 26)

Every tuple c reported by some object in its w field in the first round of the READ, is added by the reader to the set of *candidate values*, the set C (line 24). A candidate value c is automatically removed from C if at least $t + b + 1$ objects respond (in any round of the READ) without c in their w field (lines 2 and 27-28).

In the first round, the reader r_j awaits responses from a set that contains at least $S - t = t + b + 1$ objects such that there is no *conflict* between any 2 objects s_i and s_k that belong to this set (set $Resp1OK$, line 11). A conflict between two objects arises when one object, say s_k, reports in its w field a candidate value c, such that $c.tsrarray[i][j] > tsrFR$ (line 4), where $tsrFR$ is the timestamp of the reader r_j in the first round of READ (line 9). In other words, object s_k claims that the object s_i reported to the writer a timestamp of the reader r_j higher than any timestamp that r_j has issued so far. Intuitively, in this case, at least one of the objects s_k or s_i is malicious. Hence, in a set that contains only correct objects there is no conflict between any two objects. As there are at least $S - t$ correct objects, hence the intuition on why the first round of READ eventually completes (i.e., why the condition in line 11 eventually holds).

At the beginning of the second round of the READ, the reader r_j increments its local timestamp tsr'_j once more (line 12) and sends a $READ2\langle tsr'_j \rangle$ to all objects (line 13). Then the reader waits for the responses from objects until there is a candidate value c with the highest timestamp in C (i.e., $highCand(c)$ holds, line 4), such that $safe(c)$ holds or until C is empty (this can occur only if the READ is concurrent with some WRITE). The predicate $safe(c)$ holds if at least $b + 1$ different objects have responded either in their w (or pw) fields with c (or $c.tsval$ for pw), or with a value with a higher timestamp (line 3).

Our implementation guarantees that the condition in line 14 is eventually satisfied in every READ. In the following, we give a rough intuition on why this is true (the detailed proof of correctness can be found in the full paper [10]).

Assume, by contradiction, that there is a READ rd by some reader r_j (in run r) such that rd never completes, i.e., there is a candidate value c in rd, such that c is never eliminated from C and c is never *safe*. Consider the following three cases.

- Candidate value c was reported by at least one correct object in the first round of the READ rd. In this case, at least $b + 1$ correct objects have already set their pw fields to $c.tsval$ before the second round of rd is invoked and these objects reply in the second round with $c.tsval$ or a later value in their pw fields and, hence, $safe(c)$ eventually holds.

Definitions:
1: $conflict(i,k) ::= \exists c \in C :$
$((k \in FirstRW(c)) \wedge (c.tsrarray[i][j] > tsrFR))$
2: $RespondedWO(c) := \{i : \exists c' \neq c, i \in RW(c')\}$
3: $safe(c) ::= |RW(c) \cup RPW(c.tsval) \bigcup$
$\bigcup_{c'.tsval.ts > c.tsval.ts}(RW(c') \cup RPW(c'.tsval))| \geq b+1$
4: $highCand(c) ::= (c \in C) \wedge (\neg\exists c' \in C : c'.tsval.ts > c.tsval.ts)$
5: $Resp1 := \{i : RespFirst[i] = true\}$

Initialization:
6: $tsr'_j := 0$

READ() is {
7: $C := FirstRW := RW := RPW := \emptyset$
8: $RespFirst[i] := false, 1 \leq i \leq S$
9: $tsrFR := tsr'_j := tsr'_j + 1$
10: send $READ1\langle tsr'_j\rangle$ to all objects
11: **wait for** $READ1\_ACK_i$ messages **until**
$\exists Resp1OK \subseteq Resp1 :$
$(|Resp1OK| \geq S - t) \wedge (\forall i, k \in Resp1OK : \neg conflict(i,k))$
12: $inc(tsr'_j)$
13: send $READ2\langle tsr'_j\rangle$ to all objects
14: **wait for** $READ2\_ACK_i$ messages **until**
$\exists c_{ret} \in C : ((safe(c_{ret}) \wedge highCand(c_{ret})) \vee (C = \emptyset))$
15: **if** $C = \emptyset$ **then**
16: **return**(v_0)
17: **else**
18: $c_{ret} := c : ((c \in C) \wedge (safe(c)) \wedge (highCand(c)))$
19: **return**$(c_{ret}.tsval.v)$
20: **endif**

21: **upon** reception of $READ1\_ACK_i\langle tsr'_j, pw', w'\rangle$ from s_i **do**
22: $FirstRW(w') := FirstRW(w') \cup \{i\}$
23: $RW(w') := RW(w') \cup \{i\}$; $RPW(pw') := RPW(pw') \cup \{i\}$
24: $C := C \cup \{w'\}$; $RespFirst[i] := true$

25: **upon** reception of $READ2\_ACK_i\langle tsr'_j, pw', w'\rangle$ from s_i **do**
26: $RW(w') := RW(w') \cup \{i\}$; $RPW(pw') := RPW(pw') \cup \{i\}$;

27: **upon** $(c \in C)$ **and** $(|RespondedWO(c)| \geq t + b + 1)$
28: $C := C \backslash \{c\}$
}

Figure 4: SWMR safe storage: READ implementation - code of the reader r_j

- Consider now the second case, in which no correct object ever reports c in its w field to r_j. Eventually all correct objects, at least $S - t = t + b + 1$ of them respond with some value different from c in their w fields and c is excluded from C (lines 27-28).

- Finally, consider the third case, in which (1) no correct object reports c in its w field in the first round of the READ rd and (2) at least one correct object reports c in its w field in the second round of rd. In this case, some malicious objects have forged c, but c was indeed later written concurrently with the READ rd. Note that the value of the array of arrays of reader timestamps reported to the writer, $c.tsrarray$, is crucial in this case. It contains values of $tsr[j]$ fields of at least $S-t-t = b+1$ correct objects that those objects reported to the writer during the WRITE wr (concurrent with rd) that actually wrote c. Denote by $tsrFR$ the timestamp of the reader r_j in the first round of rd. Note that a correct object s_i sets $tsr[j]$ to a value higher than $tsrFR$ (i.e., $tsrFR + 1$, since by our assumption, rd never completes and, therefore, r_j never sets its timestamp to a value higher than $tsrFR + 1$) only upon s_i receives a second round message of rd.

  For every such a correct object s_i, if $c.tsrarray[i][j] \leq tsrFR$, the object s_i will respond to the second round of rd with $c.tsval$ in its pw field or with a later value (otherwise $c.tsrarray[i][j] > tsrFR$ in the PW round of wr). On the other hand, if $c.tsrarray[i][j] > tsrFR$ at the end of the first round of rd, every (malicious) object that reported c in its w field in the first round of the READ will be in conflict with s_i. Therefore, (1) at the end of the first round of READ, s_i is not in $Resp1OK$ and (2) s_i responds *without* c (and $c.tsval$) in the second round of the READ. Roughly, in our algorithm, below a certain threshold of correct objects s_i for which $c.tsrarray[i][j] > tsrFR$, $safe(c)$ will eventually hold. If the number of correct objects s_i such that $c.tsrarray[i][j] > tsrFR$ crosses this threshold, eventually the number of objects that responded without c in their w fields becomes larger than $t + b$, i.e., c is removed from C.

In other words, in any run r of our algorithm, for any $c \in C$, $safe(c)$ eventually holds in r, or c is eventually removed from C (in r).

5. REGULAR IMPLEMENTATION

Our tight lower bound on the time-complexity of READ operations extends to stronger storage semantics: optimally resilient *regular* storage. In this section, we show how to transform our safe implementation (Section 4) to provide regular semantics while retaining optimal resilience and optimal time-complexity of READ and WRITE operations (i.e., rounds). (For the lack of space, the proof of correctness of the transformation is given in the full paper [10].)

The main difference between our regular implementation and our safe implementation, is that objects keep track of all values they receive from the writer throughout the entire run (for simplicity we say that objects *store the entire history*). For presentation simplicity, we will assume in the following that in every READ round, objects send all the values received from the writer (i.e., the entire history) to the reader.

However, later, at the end of this section, we show how to simply optimize our implementation in order to drastically decrease the size of messages exchanged between objects and readers in our algorithm (as well as memory requirements and computational complexity at readers).

The communication pattern of our regular implementation is the same as that of our safe implementation of Section 4. Moreover, the principle of choosing the value to return in the reader code is essentially the same, only the set of candidate values to choose from becomes larger than in our safe implementation.

The WRITE implementation remains unchanged, i.e., we can reuse the implementation given in Figure 2, Section 4.

However, object s_i, on reception of $PW\langle ts', pw', w'\rangle$ from the writer, with $ts' > ts$, where ts is the timestamp of the latest PW or W message received by s_i from the writer, updates ts and assigns $history_i[ts'] := \langle pw', nil\rangle$ and $history_i[ts'-1] := \langle w'.tsval, w'\rangle$ (lines 5-7, Figure 5). Similarly, on reception of $W\langle ts', pw', w'\rangle$ from the writer, with $ts' \geq ts$, s_i updates ts and assigns $history_i[ts'] := \langle pw', w'\rangle$ (lines 11-12, Figure 5).

Initialization:
1: $ts := 0$; $pw_0 := \langle 0, \perp\rangle$; $history_i[0] := \langle pw_0, \langle pw_0, inittsrarray\rangle\rangle$
2: $inittsrarray[i][j] := nil$, $1 \leq i \leq S$, $1 \leq j \leq R$
3: $tsr[j] := 0$, $1 \leq j \leq R$

4: **upon** reception of $PW\langle ts', pw', w'\rangle$ message from the writer **do**
5: **if** $ts' > ts$ **then**
6: $history_i[ts] := \langle pw', nil\rangle$; $history_i[ts-1] := \langle w'.tsval, w'\rangle$
7: $ts := ts'$
8: send $PW\_ACK_i\langle ts, tsr\rangle$ to the writer
9: **endif**

10: **upon** reception of $W\langle ts', pw', w'\rangle$ message from the writer **do**
11: **if** $ts' \geq ts$ **then**
12: $ts := ts'$; $history_i[ts] := \langle pw', w'\rangle$
13: send $WRITE\_ACK_i\langle ts\rangle$ to the writer
14: **endif**

15: **upon** reception of $READk\langle tsr'\rangle$ mess. from r_j ($k \in \{1, 2\}$) **do**
16: **if** $tsr' > tsr[j]$ **then**
17: $tsr[j] := tsr'$
18: send $READk\_ACK_i\langle tsr[j], history_i\rangle$ to the reader r_j
19: **endif**

Figure 5: SWMR regular storage: code of object s_i

Moreover, on reception of the $READk$ message from the reader with a timestamp tsr', the object s_i replies with the message $READk\_ACK_i\langle tsr', history_i\rangle$, where k denotes the round ($k \in \{1, 2\}$). (We later show how the size of $READk\_ACK_*$ messages can be drastically decreased). The entire modified object code is given in Figure 5.

We give the modified reader code in Figure 6. The reader r_j, on receiving the $READk\_ACK_i\langle tsr', history_i\rangle$ message from object s_i in round k of READ rd, assigns $history[k][i] := history_i$ (line 19 and 24, Fig. 6). If, for some ts' the entry $history_i[ts']$ does not exist, r_j considers $history[k][i][ts'] = history_i[ts'] = \langle nil, nil\rangle$. The reader adds (non-nil) values of tuples $history[1][i][*].w$, i.e., the values objects report in their $history_i[*].w$ fields, into the set of candidate values C throughout the first round of rd (line 20, Fig. 6)

Similarly to our safe implementation, in the first round of rd, the reader r_j awaits responses from a set that contains at least $S - t$ objects such that there is no *conflict* between any 2 objects s_i and s_k that belong to this set (line 11, Fig. 6). A conflict between two objects arises when

Definitions:
1: $conflict(i, k) ::= \exists c \in C, \exists ts'$: $(history[1][k][ts'].w = c) \wedge (c.tsrarray[i][j] > tsrFR))$
2: $invalid(c) ::= |\{i : \exists rnd \in \{1, 2\}$: $(history[rnd][i][c.tsval.ts].w = nil) \vee$ $\vee (history[rnd][i][c.tsval.ts].pw \neq c.tsval) \vee$ $\vee (history[rnd][i][c.tsval.ts].w \neq c)\}| \geq t + b + 1$
3: $safe(c) ::= |\{i : \exists rnd \in \{1, 2\}$: $(history[rnd][i][c.tsval.ts].pw = c.tsval) \vee$ $\vee (history[rnd][i][c.tsval.ts].w = c)\}| \geq b + 1$
4: $highCand(c) ::= (c \in C) \wedge (\neg\exists c' \in C : c'.tsval.ts > c.tsval.ts)$
5: $Resp1 ::= \{i : RespFirst[i] = true\}$

Initialization:
6: $tsr'_j := 0$

READ() is {
7: $history[1..2][1..S] := init$
8: $tsr[i] := 0$; $RespFirst[i] := false, 1 \leq i \leq S$
9: $tsrFR := tsr'_j := tsr'_j + 1$
10: send $READ1\langle tsr'_j\rangle$ to all objects
11: **wait for** $READ1\_ACK_i$ messages **until** $\exists Resp1OK \subseteq Resp1$: $(|Resp1OK| \geq S-t) \wedge (\forall i, k \in Resp1OK : \neg conflict(i, k)))$
12: $inc(tsr'_j)$
13: send $READ2\langle tsr'_j\rangle$ to all objects
14: **wait for** $READ2\_ACK_i$ messages **until** $\exists c_{ret} \in C : ((safe(c_{ret}) \wedge (highCand(c_{ret})))$
15: $c_{ret} := c : (c \in C) \wedge safe(c) \wedge (highCand(c))$
16: **return**$(c_{ret}.tsval.v)$

17: **upon** reception of $READ1\_ACK_i\langle tsr'_j, history_i\rangle$ from s_i **do**
18: **if** $(tsr'_j > tsr[i])$ **then**
19: $tsr[i] := tsr'_j$; $history[1][i] := history_i$
20: $C := C \cup \{history_i[*].w'\}$; $RespFirst[i] := true$
21: **endif**

22: **upon** reception of $READ2\_ACK_i\langle tsr'_j, pw', w'\rangle$ from s_i **do**
23: **if** $(tsr'_j > tsr[i])$ **then**
24: $tsr[i] := tsr'_j$; $history[2][i] := history_i$
25: **endif**

26: **upon** $(c \in C)$ **and** $(invalid(c))$
27: $C := C\backslash\{c\}$
}

Figure 6: SWMR regular storage: READ implementation - code of the reader r_j

one object, say s_k reports (in the first round of read) in one of its $history_k[*].w$ fields a candidate value c, such that $c.tsrarray[i][j] > tsrFR$, where $tsrFR$ is the timestamp of the reader r_j in the first round of the READ. As in our safe implementations, there can be no conflict between two correct objects s_i and s_k.

We define two key predicates for candidate values $c \in C$, $safe(c)$ and $invalid(c)$ as follows:

- $safe(c)$. A candidate value c is *safe* if at least $b + 1$ objects s_i have responded with either $c.tsval$ or c in the pw or w field (respectively) of $history_i[c.tsval.ts]$ in either the first, or the second round of the READ. (line 3, Fig.6). In other words, c if safe if at least $b + 1$ objects confirm that the timestamp-value pair $c.tsval$ has been written by the writer in a write with a timestamp $c.tsval.ts$.

- $invalid(c)$. A candidate value c is deemed *invalid* if at least $t + b + 1$ objects s_i are missing the entry $history_i[c.tsval.ts].w$ (i.e., if $history_i[c.tsval.ts].w = nil$), or reply with a value different than $c.tsval$ (resp., c) in the pw (resp., w) field of their $history_i[c.tsval.ts]$, in either the first, or the second round of READ. (line 2, Fig.6). In other words, c if invalid if at least $t+b+1$ objects did not receive c with a timestamp $c.tsval.ts$ from the writer.

As soon as the predicate $invalid(c)$ holds, c is removed from the set C (lines 26 and 27, Fig. 6).

The reader receives $READ2\_ACK_i$ messages (in the second round of READ) until there is a candidate value c such that $safe(c)$ holds and there is no other candidate value with a higher timestamp. This is guaranteed to occur at latest after the reader receives the responses from all correct objects in the second round of READ. Roughly, the principle behind this fact, is the same as in our safe implementation.

It is relatively easy to see how we can simply modify our regular implementation such that objects do not send their entire histories to readers within the $READk\_ACK_i$ messages. Consider READ rd by r_j. It is sufficient that the reader r_j stores (caches) the value $cache_j.val$ it returned in its last READ that preceded rd along with the timestamp associated with $cache_j.val$, $cache_j.ts$. Then, in the first round of rd, r_j includes $cache_j.ts$ in its $READ1$ message, and the object s_i send in $READk\_ACK_i$ messages in rd only the portion of the $history_i$ from $history_i[cache_j.ts]$ onwards. It may occur in this case that, after two rounds of READ, the set C is empty. In this case, r_j simply returns $cache_j.val$. The rest of the algorithm can be reused as such.

6. SERVER-CENTRIC MODEL

We extend our model of Section 2 to a server-centric model, by assuming point-to-point channels among objects (servers) and removing the restriction that objects can send messages only in response to clients. In other words, in the server-centric model, base objects are first class processes (servers) that can exchange messages with other servers and even send unsolicited messages to clients (i.e., *push* messages). As a consequence, the range of communication patterns is very broad and not bound by the pattern of a communication round-trip. Clearly, the notion of a communication round-trip needs to be revisited as a complexity metric.

For example, clients in a server-centric model may send only one message to (a subset of) servers and wait for the reception of pushed messages, until they receive sufficient amount of information for returning a value. It is not difficult to see that, in an asynchronous system, clients need only to send this first message to a subset of servers in order to return a meaningful value.

The notion of a single communication round-trip (round) and fast READ operations (that complete in a single round) is however meaningful even in the server centric model [7]. Intuitively, a fastest possible operation in this model is similar to that of our data-centric model; i.e., a fast operation *op* in which: (a) the client c sends messages to (a subset of) servers, (b) servers, on receiving such a message, reply to c, *without waiting for the reception of any other message from any other server or client* and (c) upon c receiving a sufficient number of these replies (at latest upon c receives replies from $S - t$ correct servers) *op* completes.

It is not difficult to see, along with our lower bound proof of Section 3, that our lower bound (Proposition 1 of Section 3) holds in the server-centric model as well (with the fast READ operations defined as above). In other words, even in the server-centric model, if at most $2t + 2b$ servers are used, then it is impossible to construct a SWSR safe regular storage in which every READ is fast. Devising a tight bound algorithm, however, might require a different metric; this is however out of the scope of this paper.

Acknowledgments

We are very thankful to Gregory Chockler, Idith Keidar and Ron Levy for their very helpful comments on earlier drafts of this paper.

7. REFERENCES

[1] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 226–235. ACM Press, 2004. Full version to appear in *Distributed Computing*.

[2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Wait-free regular storage from Byzantine components. Technical Report MIT-CSAIL-TR-2005-021, MIT, Cambridge, MA, USA, 2005.

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

[4] R. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of the 18th International Symposium on Distributed Computing*, volume 3274/2004 of *Lecture Nodes in Computer Science*, pages 405–419, Oct 2004.

[5] R. A. Bazzi and Y. Ding. Brief announcement: wait-free implementation of multiple-writers/multiple-readers atomic byzantine data storage systems. In M. K. Aguilera and J. Aspnes, editors, *PODC*, page 353. ACM, 2005.

[6] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. Technical Report RZ 3575, IBM Research, February 2005.

[7] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolić. How Fast can a Distributed Atomic Read be?

Technical Report LPD-REPORT-2005-001, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, 2005. Preliminary version appeared in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, 2004.

[8] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, 2004.

[9] R. Guerraoui, R. R. Levy, and M. Vukolić. Lucky read/write access to robust atomic storage. Technical Report LPD-REPORT-2005-005, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, 2005. Preliminary version appeared in *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[10] R. Guerraoui and M. Vukolić. How fast can a very robust read be? Technical Report LPD-REPORT-2006-008, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, May 2006.

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

[12] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

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

[14] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[15] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

[16] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, 1998.

[17] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[18] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325. Springer-Verlag, 2002.

[19] M. K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 68–80. ACM Press, 1994.

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