Lucky Read/Write Access to Robust Atomic Storage

Rachid Guerraoui[1,2], Ron R. Levy[1] and Marko Vukolić[1]
[1]*School of Computer and Communication Sciences, EPFL, CH-1015 Lausanne, Switzerland*
[2]*Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA*
{*rachid.guerraoui, ron.levy, marko.vukolic*}*@epfl.ch*

Abstract

This paper establishes tight bounds on the best-case *time-complexity of distributed atomic read/write storage implementations that tolerate* worst-case *conditions. We study asynchronous robust implementations where a writer and a set of reader processes (clients) access an atomic storage implemented over a set of* $2t+b+1$ *server processes of which* t *can fail:* b *of these can be malicious and the rest can crash. We define a* lucky *operation (read or write) as one that runs synchronously and without contention. It is often argued in practice that lucky operations are the most frequent. We determine the exact conditions under which a* lucky *operation can be* fast*, namely expedited in one-communication round-trip with no data authentication. We show that every* lucky *write (resp., read) can be* fast *despite* f_w *(resp.,* f_r*) actual failures, if and only if* $f_w + f_r \leq t - b$*.*

1 Introduction

It is considered good practice to *plan for the worst and hope for the best*. This practice has in particular governed many complexity studies in dependable distributed computing [9, 18, 10].

It is indeed admitted that distributed algorithms ought to tolerate bad conditions with many processes failing or competing for shared resources such as communication channels. Under these conditions, the distributed system is clearly asynchronous as there is no realistic bound on relative process speeds and communication delays.

Fortunately, those bad conditions are considered rare and whilst it is good to make sure algorithms tolerate them when they happen, one would rather optimize the algorithms for the common, not that bad conditions. It is for instance often argued that distributed systems are synchronous most of the time, that there is generally little contention on the same resources at any given point in time [4] and that failures are rare. Hence, when measuring the complexity of a distributed algorithm and judging whether the algorithm is *fast*, it is reasonable to measure its (time) complexity under such best case conditions. An algorithm that would be efficient under rare worst case conditions and slow under frequent best case conditions, would not be practically appealing.

In this paper, we study distributed algorithms that implement the classical single-writer multi-reader (SWMR) *atomic storage* abstraction [17]: a fundamental notion in dependable distributed computing [20, 3]. This abstraction, also called the SWMR *atomic register*, captures the semantics of a shared variable on which processes, called clients, can concurrently invoke read and write operations. Atomicity stipulates that (1) a read r must return a value written by a concurrent write (if any), or the last value written, and (2) if a read r' that precedes r returns value v, then r must return either v or a later value.

We consider robust, sometimes called wait-free, implementations of an atomic storage where no client relies on any other client to complete any of its operations: the other clients might have all stopped their computation (crashed) [17, 14, 2]. The storage abstraction is implemented over a set of $2t + b + 1$ server processes of which t can fail: b of these can be malicious, deviating arbitrarily from the algorithm assigned to them, while the rest can crash. In this paper, we assume that stored data is not authenticated.[1]

It is a known result that $2t + b + 1$ is a resilience lower bound for any safe [17] storage implementation in an asynchronous system [21][2] and, to ensure atomicity with this resilience, more than one communication-round trip is needed between a client (the writer or the readers) and the servers [1, 11]. In [2] for instance, even if only server crashes are tolerated (i.e., $b = 0$), the reader needs to send a message to all servers, wait for their replies, determine the latest value, send another message to all servers, wait for

[1]For completeness, we consider the impact of using data authentication in Section 5.

[2]Actually, [21] proves the optimal resilience lower bound for the special case where $b = t$. It is not difficult to extend this result for $b \neq t$ using the same technique.

replies, and then return the value. A total of two communication round-trips is thus required for every read operation.

The goal of this paper is to determine the exact conditions under which optimally resilient implementations that tolerate asynchrony and contention (worst-case conditions), can expedite operations (reads or writes) whenever the system is synchronous and there is no contention. We say that an operation is *lucky* if (a) it runs synchronously and (b) without contention. In short, this means that (a) the client that invokes the operation reaches and receives replies from all non-faulty servers in one communication round-trip, and (b) no other client is invoking a conflicting operation (no read is overlapping with a write).

We define the notion of a *fast* operation as one that executes in one communication round-trip between a client and the servers. Indeed, it is usual to measure the time complexity of an operation by counting the number of communication rounds needed to complete that operation, irrespective of the local computation complexity at any process involved in the operation (server or client). The rationale behind this measure is that local computation is negligible with respect to communication rounds (assuming data is authenticated).

This paper shows that in order for every *lucky* write to be *fast*, despite at most f_w actual server failures, and every *lucky* read to be *fast*, despite at most f_r actual server failures, it is *necessary* and *sufficient* that the sum $f_w + f_r$ is not greater than $t - b$. This result expresses the precise tradeoff between the thresholds f_w and f_r.

We proceed as follows. We first give an algorithm with $2t + b + 1$ servers that tolerates asynchrony and t server failures among which b can be malicious: the algorithm allows every *lucky* read (resp., *lucky* write) to be *fast* in any execution where up to f_w (resp., f_r) servers fail, provided $f_w + f_r = t - b$. Note that all f_w (resp. f_r) failures can be malicious, provided $f_w \leq b$ (resp. $f_r \leq b$). The challenge underlying the design of our algorithm is the ability to switch to slower operations that preserve atomicity under the worst conditions: asynchrony and contention, as well as t failures out of which b can be malicious, with $2t+b+1$ servers (optimal resilience [21]). A key component of our algorithm is a signalling mechanism we call "freezing", used by the reader to inform the writer of the presence of contention. Interestingly, this mechanism does not rely on intercommunication among servers, nor on servers pushing messages to the clients.

We then give our matching upper bound result by showing that no optimally resilient asynchronous algorithm can have every *lucky* write be *fast* despite f_w actual server failures and every *lucky* read be *fast* despite f_r actual server failures, if $f_w + f_r > t - b$. Our upper bound proof is based on indistinguishability arguments that exploit system asynchrony, the possibility of some servers to return an arbitrary value, and the requirement that every write wr (resp., read rd) must be *fast* whenever wr (resp., rd) is *lucky* and at most f_w (resp., f_r) servers are faulty. To strengthen our tight bound, we assume in our proof a general model in which servers can exchange messages and even send unsolicited messages.

We use the very fact that our upper bound proof requires *every lucky* operation (in particular, every *lucky* read) to be *fast*, to (1) drastically increase the sum of the thresholds f_w and f_r to $f_w = t - b$ and $f_r = t$, by allowing a certain number, yet just a small fraction, of *lucky* read operations to be *slow*. We also highlight the fact that (2) our upper bound is inherent to atomic storage implementations, but does not apply to weaker semantics; we discuss how to modify our algorithm and get a regular [17] storage implementation in which every *lucky* write (resp., read) is *fast* despite the failure of $f_w = t - b$ (resp., $f_r = t$) servers. We prove the optimality of (1) and (2) by showing, along the lines of [1], that no optimally resilient safe [17] algorithm can achieve *fast* writes despite the failure of more than $t - b$ servers.

The rest of the paper is organized as follows. We present in Section 2 our general system model together with few definitions that are used in the rest of the paper. In Section 3 we present our algorithm, which we prove optimal in Section 4. In Section 5 we discuss several alternatives to the assumptions underlying our result. We conclude the paper by discussing the related work in Section 6.

2 System Model and Definitions

The distributed system we consider consists of three *disjoint* sets of processes: a set *servers* of size S containing processes $\{s_1, ..., s_S\}$, a singleton *writer* containing a single process $\{w\}$, and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. We denote a set of *clients* as a union of the sets *writer* and *readers*. We assume that every client may communicate with any server by message passing using point-to-point reliable communication channels. When presenting our algorithm, we assume that servers send messages only in reply to a message received from a client: i.e., servers do not communicate among each other, nor send unsolicited messages. However, to strengthen our tight upper bound, we relax these assumptions. To simplify the presentation, we assume a global clock, which, however, is not accessible to either clients or servers.

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. A distributed algorithm A is a collection of automata. 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 $\langle p, M\rangle$ (M might be $\emptyset$). In step $sp = \langle p, M\rangle$, 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 $mset_{p,*}$. 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. Note that the malicious process p cannot remove/put any message into a point-to-point channel between any two non-malicious processes q and r.

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 $\langle p, M\rangle$ 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 $\langle p, M\rangle$ 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 $\langle p, M\rangle$ 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 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, at most t servers might be faulty, out of which at most $b \leq t$ may be *malicious*. In this paper we consider only optimally resilient implementations [21], where the total number of servers S equals $2t + b + 1$.

For presentation simplicity, we do not explicitly model the initial state of a process, nor the invocations and responses of the read/write 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 response).

2.2 Atomic Register

A sequential (read/write) storage is a data structure accessed by a single process. It provides two operations: WRITE(v), which stores v in the storage, and READ(), which returns the last value stored. An atomic storage is a distributed data structure that may be concurrently accessed by multiple clients and yet provides an "illusion" of a sequential storage to the accessing clients.

We refer the readers to [17, 20, 14, 15] for a formal definition of an atomic storage, and we simply recall below what is required to state and prove our results.

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 one). 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. 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 $op2$ precede the other, the operations are said to be *concurrent*.

An algorithm *implements* a robust atomic storage if every run of the algorithm satisfies *wait-freedom* and *atomicity* 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 atomicity for the SWMR atomic storage.

In the single-writer setting, WRITEs in a run have a natural ordering which corresponds to their physical order. Denote by wr_k the k^{th} WRITE in a run ($k \geq 1$), and by val_k the value written by the k^{th} WRITE. Let $val_0 = \perp$. We say that a partial run satisfies atomicity if the following properties hold: (1) if a READ returns x then there is k such that $val_k = x$, (2) if a READ rd is complete and it succeeds some WRITE wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$, (3) if a READ rd returns val_k ($k \geq 1$), then wr_k either precedes rd or is concurrent to rd, and (4) if some READ $rd1$ returns val_k ($k \geq 0$) and a READ $rd2$ that succeeds $rd1$ returns val_l, then $l \geq k$.

2.3 Lucky Operations

A complete READ/WRITE operation op by the client c is called *synchronous*, if the message propagation time for every message m exchanged in time period $[t_{op_{inv}}, t_{op_{resp}}]$, where op is invoked at $t_{op_{inv}}$ and completed at time $t_{op_{resp}}$, between client c and any server s_i is bounded by the constant t_{c,s_i} known to the client c. A complete operation op is *contention-free* if it is not concurrent with any other WRITE wr. An operation op is *lucky* if it is synchronous and contention-free. Note that, in our SWMR setting, every *synchronous* WRITE operation is *lucky*.

2.4 Fast Operations

Basically, we say that a complete operation op is *fast* if op completes in one communication round; otherwise, op is *slow*. In other words, in a *fast* READ (resp., WRITE):

1. The reader (resp., writer) sends messages to a subset of servers in the system (possibly all servers).

2. Servers on receiving such a message reply to the reader (resp., writer) before receiving any other messages. More precisely, any server s_i on receiving a message m in step $sp1 = \langle s_i, M \rangle$ ($m \in M$), where m is sent by the reader (resp., writer) on invoking a READ (resp., WRITE), replies to m either in step $sp1$ itself, or in a subsequent step $sp2$, such that s_i does not receive any message in any step between $sp1$ and $sp2$ (including $sp2$). Intuitively, this requirement forbids the server to wait for some other message before replying to m.

3. upon the reader (resp., writer) receives a sufficient number k of such replies, a READ (resp., WRITE) completes.

3 Algorithm

Proposition 1. There is an optimally resilient implementation I of a SWMR robust atomic storage, such that: (1) in any partial run in which at most f_w servers fail, every *lucky* WRITE operation is *fast*, and (2) in any partial run in which at most f_r servers fail, every *lucky* READ operation is *fast*, where $f_w + f_r = t - b$.

In the following, we first give an overview of the algorithm and then we give a detailed description of the WRITE and READ implementations.[3]

3.1 Overview

If a WRITE is synchronous (i.e., *lucky*) and at most f_w servers are faulty, the WRITE is *fast* and completes in a single (communication) round. A *slow* WRITE takes an additional two rounds. The READ operation also proceeds in series of rounds (a *fast* READ, completes in a single round). In every round, a client sends a message to all servers and awaits a response from $S - t$ different servers. In addition, in the first round of every operation, a client c awaits responses until the expiration of the timer, set according to the message propagation bounds t_{c,s_*} (see Section 2.3).

Roughly speaking, a *fast* WRITE, writes the new value in at least $S - f_w$ servers. Consider a *lucky* READ rd, such that the last WRITE that precedes rd is a *fast* WRITE wr that writes v_{fast} and that at most f_r servers are faulty. In this case, v_{fast} is written into at least $S - f_w$ servers, out of which a set X containing at least $S - f_w - f_r = 2b + t + 1$ servers are correct. Since no value later than v_{fast} is written before rd completes (since rd is *lucky*), all servers from the set X will respond with v_{fast} in the first round of rd. Similarly, if wr is a *slow* WRITE, in every (out of three) rounds, wr writes v_{slow} to at least $S - t$ servers. In this case, a *lucky* READ rd that comes after wr will read, in its first round, a value v_{slow} written in the third (final) round of wr from at least $S - t - f_r \geq b + 1$ correct servers. In both cases, our algorithm guarantees that rd is *fast* and that it returns v_{fast} (resp., v_{slow}) at the end of the first round.

However, since rd is *fast*, it does not send any additional messages to servers after the first round. Therefore, when returning a value v, a *fast* READ rd must itself "leave" behind enough information so the subsequent READS will not return the older value. This is precisely the case, when rd encounters a set X containing at least $2b + t + 1$ (resp., $b + 1$) servers that "witness" a *fast* (resp., *slow*) WRITE. To illustrate this, consider a READ rd' by some reader r_j that succeeds rd. In addition, for simplicity, assume that no WRITE operation that succeeds wr is invoked (naturally, the correctness of our algorithm does not rely on this assumption). In case wr is *fast*, r_j is guaranteed to receive a response from at least $2b + t + 1 - (t + b) = b + 1$ servers that belong to the set X, in every round of rd', overwhelming the number of responses from malicious servers (at most b) that may be trying to mislead r_j. Now consider the case where wr is *slow*. Out of at least $b+1$ servers that "witness" a third (final) round of wr and respond to a *fast* READ rd, at least one is non-malicious, which means that the second round of the write wr completed and a set Y containing at least $S - t - b = t + 1$ non-malicious servers "witnessed" the second round of wr. Reader r_j is guaranteed to receive a response from at least one non-malicious server s_i, that belongs to the set Y in every round of rd'. Roughly speaking, s_i claims that the first round of wr completed and that a set Z of at least $S - t - t = b + 1$ correct servers "witnessed" the first round of wr. All the servers from set Z will eventually respond to rd' confirming the claims of s_i.

A value v is returned only if at least $b + 1$ servers report the exact value v. Since servers do not store the entire history of all the values they receive, in the case the writer issues an unbounded number of WRITEs and if readers do not inform the writer about their (*slow*) READs, server data can repeatedly be overwritten. This leads to the impossibility of confirming any value at $b + 1$ servers. To solve this issue, our algorithm employs a careful signalling between the readers and the writer, a mechanism we call *freezing*. Roughly, to initiate *freezing*, a *slow* READ rd' by reader r_j writes its own timestamp ts'_{r_j} to all servers. Every server s_i appends ts'_{r_j} to its reply to the first round message of every

[3] For space limitations, the correctness proof is omitted; for details see the full paper [13].

subsequent WRITE (until the writer "freezes" the value for rd'). As soon as the writer receives ts'_{r_j} from at least $b+1$ different servers, the writer "freezes" the value for rd' and writes it in the dedicated server field, $frozen_{r_j}$. Our algorithm guarantees that the writer "freezes" at most one value per (*slow*) READ. The READ rd' reads the servers' value of $frozen_{r_j}$ and is guaranteed to eventually return a value.

Finally, a *slow* READ writes back the value v it returns in a well-known manner [2]. The writeback procedure follows the communication pattern of the WRITE operation and, hence, takes three communication rounds.

3.2 WRITE implementation

The pseudocode of the WRITE implementation is given in Figure 1. The writer maintains the following local variables: (1) a local timestamp ts initially set to ts_0, (2) a timestamp-value pairs pw and w initially set to $\langle ts_0, \perp \rangle$, (3) array $read\_ts[*]$ initially set to $read\_ts[r_j] = \langle tsr_0 \rangle$, for every reader r_j, where tsr_0 is the initial local timestamp at every reader (see Section 3.3).

The WRITE operation consists of two phases: pre-write (PW) phase and write (W) phase. The writer w begins the PW phase of operation $wr = WRITE(v)$ by increasing its local timestamp ts, updating its pw variable to reflect the new timestamp-value pair $\langle ts, v \rangle$ and triggering the timer T (line 3). Then, the writer sends the $PW\langle ts, pw, w, frozen \rangle$ message to all servers (line 4). The field $frozen$ of the PW message is sent optionally in case the writer has to "freeze" a value for some ongoing READ. On reception of a $PW\langle ts, pw', w', * \rangle$ message, every server updates its local copy of pw and w, if these are older than pw' and w', respectively. Even if $PW.pw'$ and $PW.w'$ are older than the servers' local copies pw and w, servers take into account the information in the $frozen$ field of the PW message (lines 5-6, Fig. 3). Servers reply to the writer with a $PW\_ACK\langle ts, newread \rangle$ message. In the optional $newread$ field, servers inform the writer about the *slow* READS that have difficulties returning a value.

In the PW phase, the writer awaits both for valid responses [4] to the PW message from $S-t$ different servers and the expiration of the timer. The writer completes the PW phase by executing the $freezevalues()$ procedure, that consists of local computations only (line 7). If the writer received at least $S - f_w$ valid PW_ACK messages, the WRITE completes. Otherwise, the writer proceeds to the second, W phase.

The $freezevalues()$ procedure detects ongoing *slow* READS. Namely, in every READ invocation, the reader r_j increases its local timestamp tsr_j and, unless the READ is *fast*, r_j stores this timestamp into servers' variable ts_{r_j}. In every round of WRITE operation, servers piggyback those timestamps along the id of the issuing reader to PW_ACK message within the $newread$ field, in case the writer did not already "freeze" a value for this READ (lines 5-7 Fig. 3). When the writer detects $b+1$ servers that report a timestamp for the reader r_j (ts_{r_j}) higher than the writer's locally stored value of $read\_ts[r_j]$ (line 13, Fig. 1), the writer updates its $read\_ts[r_j]$ value to the $b+1^{st}$ highest ts_{r_j} value received in the $newread$ fields of the valid responses to the PW message (line 14, Fig. 1) and "freezes" the current value of the timestamp value pair pw for the reader r_j, by assigning $frozen := frozen \cup \langle r_j, pw, read\_ts[r_j] \rangle$. The set $frozen$ is sent to all servers within the PW message of the next WRITE invocation. On reception of a PW message with a non-empty field $frozen$, servers update their local variables $frozen_{r_j}$ if the frozen value matches the timestamp ts_{r_j} stored at the server, or if it is newer (line 6, Fig. 3). The $freezevalues()$ procedure ensures wait-freedom in runs in which the writer issues an unbounded number of WRITEs.

The write (W) phase takes two rounds. Pseudocode of W phase is depicted in lines 9-12, Fig. 1 and lines 12-17, Fig. 3. In each of the rounds, the writer sends the $W\langle round, ts, pw \rangle$ message to all servers and awaits $S-t$ valid responses from different servers. At the end of the second round of W phase, WRITE completes.

```
Initialization:
1:  pw := w := ⟨ts_0, ⊥⟩; ts := ts_0; T := timer(); frozen := ∅;
2:  ∀r_j | r_j ∈ readers : read_ts[r_j] := tsr_0

WRITE(v) is {
3:  inc(ts); pw := ⟨ts, v⟩; trigger(T)                    % pre-write (PW) phase
4:  send PW⟨ts, pw, w, frozen⟩ to all servers
5:  wait for PW_ACK_i⟨ts, newread⟩ from S − t servers and expired(T)
6:  frozen := ∅; w := ⟨ts, v⟩
7:  freezevalues()
8:  if PW_ACK_i⟨ts, *⟩ received from S − f_w servers then return(OK)
9:   else for round= 2 to 3 do                           % write (W) phase
10:     send W⟨round, ts, pw⟩ to all servers
11:     wait for reception of WRITE_ACK_i⟨round, ts⟩ from S − t servers
12:  return(OK) }

freezevalues() is {
13:  (∀r_j, |{i : (PW_ACK_i.ts = ts) ∧ ((⟨r_j, ts_{r_j}⟩ ∈
       ∈ PW_ACK_i.newread) ∧ (ts_{r_j} > read_ts[r_j]))}| ≥ b + 1 do
14:    read_ts[r_j] := b + 1^st highest value ts_{r_j}
15:    frozen := frozen ∪ ⟨r_j, pw, read_ts[r_j]⟩ }
```

Figure 1. WRITE implementation (writer)

3.3 READ implementation

The pseudocode of our READ implementation is given in Figure 2. At the beginning of every READ operation, the reader r_j increases its local timestamp tsr_j. The reader r_j proceeds by repeatedly invoking rounds (until it can safely return a value) that consist of: (1) reading the latest values of the server variables pw, w, vw and $frozen_{r_j}$ and (2)

[4] A valid response to a $PW\langle ts, *, *, * \rangle$ message is a $PW\_ACK\langle ts, * \rangle$ message, with the same ts.

writing the timestamp tsr_j to variable ts_{r_j} at every server. In every round r_j awaits $S - t$ server responses. The first round of every READ is specific: (1) r_j does not write tsr_j to servers and (2) r_j waits for both at least $S - t$ responses and for the timer triggered at the beginning of the round to expire.

Definitions and Initialization:
1: $readLive(c, i) ::= (pw_i = c) \vee (w_i = c)$
2: $readFrozen(c, i) ::= (frozen_i.pw = c) \wedge (frozen_i.tsr = tsr)$
3: $safe(c) ::= |\{i : readLive(c, i)\}| \geq b + 1$
4: $safeFrozen(c) ::= |\{i : readFrozen(c, i)\}| \geq b + 1$
5: $fastpw(c) ::= (|\{i : pw_i = c\}| \geq 2b + t + 1)$
6: $fastvw(c) ::= (|\{i : vw_i = c\}| \geq b + 1)$
7: $fast(c) ::= fastpw(c) \vee fastvw(c)$
8: $invalidw(c) ::= |\{i : \exists c' : readLive(c', i) \wedge$
$\wedge(c'ts < c.ts \vee (c'.ts = c.ts \wedge c'.v \neq c.v))\}| \geq S - t$
9: $invalidpw(c) ::= |\{i : \exists c' : pw[i] = c' \wedge$
$\wedge(c'ts < c.ts \vee (c'.ts = c.ts \wedge c'.v \neq c.v))\}| \geq S - b - t$
10: $highCand(c) ::= \forall c' \forall i : (readLive(c', i) \wedge c'.ts \geq c.ts \wedge c' \neq c) \Rightarrow$
$\Rightarrow invalidw(c') \wedge invalidpw(c')$
11: $tsr := tsr_0; T := timer();$

READ() is {
12: **inc**(tsr);
13: $rnd := 0; pw_i := w_i := \langle ts_0, v_0 \rangle, rnd_i := 0, 1 \leq i \leq S;$
14: **repeat**
15: **inc**(rnd); **if** $rnd = 1$ **then trigger**(T)
16: send $READ\langle tsr, rnd \rangle$ to all servers
17: **wait for** $READ\_ACK_i\langle tsr, rnd, *, *, *, * \rangle$ from $S - t$ servers **and**
and (**expired**(T) **or** $rnd > 1$)
18: $C := \{c : (safe(c)$ **and** $highCand(c))$ **or** $safeFrozen(c)\}$
19: **until** $C \neq \emptyset$
20: $c_{sel} := (c.val : c \in C) \wedge (\neg \exists c' \in C : c'.ts > c.ts)$
21: **if** $(\neg fast(c)$ **or** $(rnd > 1))$ **then** *writeback*(c_{sel})
22: **return**($c_{sel}.val$)

23: **upon** receive $READ\_ACK_i\langle tsr, rnd', pw', w', vw', frozen'_j \rangle$ from s_i
24: **if** $(rnd' > rnd_i)$ **then**
25: $rnd_i := rnd'; pw_i := pw'; w_i := w';$
$vw_i := vw'; frozen_i := frozen'_j$ }

writeback(c) is {
26: **for** round= 1 to 3 **do**
27: send $W\langle round, tsr, c \rangle$ message to all servers
28: **wait for** receive $WRITE\_ACK_i\langle round, tsr \rangle$ from $S - t$ servers }

Figure 2. READ implementation (reader r_j)

A reader r_j maintains the following local variables: (1) arrays $pw_i, w_i, vw_i, frozen_i, 1 \leq i \leq S$ that keep the latest copy of the server's s_i variables pw, w, vw and $frozen_{r_j}$ and (2) a local timestamp tsr_j, that is increased once at the beginning of every READ invocation. We define the predicates $readLive(c, i)$ and $readFrozen(c, i)$ in lines 1 and 2 of Figure 2, respectively, to denote that: (1) a timestamp-value pair c is seen in the latest copy of either the variable pw_i, or the variable w_i of the server s_i ($readLive(c, i)$), and (2) a timestamp-value pair c is seen in the latest copy of the $frozen_{r_j}.pw$ of the server s_i, under the condition that the last value of $frozen_{r_j}.tsr$ of the server s_i is tsr_j, the current local READ timestamp ($readFrozen(c, i)$).

A reader r_j can only return a value $c.val$ if a timestamp-value pair c is $safe$ or $safeFrozen$ (lines 3,4 and 18, Fig. 2), i.e., c must have been either (a) $readLive(c, *)$ (for $safe$) or (b) $readFrozen(c, *)$ (for $safeFrozen$),

Initialization:
1: $pw, w, vw := \langle ts_0, \perp \rangle; newread := \emptyset; ; ts_{r_j} := tsr_0$
2: $\forall r_j | r_j \in readers : \langle frozen_{r_j}.pw, frozen_{r_j}.tsr \rangle := \langle \langle ts_0, \perp \rangle, tsr_0 \rangle$

3: **upon** receive $PW\langle ts, pw', w', frozen \rangle$ from the writer
4: update(pw, pw'); update(w, w')
5: $\forall j : \langle r_j, pw'_j, tsr'_j \rangle \in frozen$ **do**
6: **if** $tsr'_j \geq ts_{r_j}$ **then** $\langle frozen_{r_j}.pw, frozen_{r_j}.tsr \rangle := \langle pw'_j, tsr'_j \rangle$
7: $newread := \bigcup \langle r_j, ts_{r_j} \rangle$, for all r_j such that $ts_{r_j} > frozen_{r_j}.tsr$
8: send $PW\_ACK_i\langle ts, newread \rangle$ to the writer

9: **upon** receive $READ\langle tsr', rnd' \rangle$ from reader r_j
10: **if** $(tsr' > ts_{r_j})$ **and** $(rnd' > 1)$ **then** $ts_{r_j} := tsr$
11: send $READ\_ACK_i\langle tsr', rnd', pw, w, vw, frozen_{r_j} \rangle$ to r_j

12: **upon** receive $W\langle round, ts, c \rangle$ from client $clnt$
13: update(pw, c)
14: **if** $round > 1$ **then** update(w, c)
15: **if** $round > 2$ **then** update(vw, c)
16: send $WRITE\_ACK_i\langle round, ts \rangle$ to $clnt$

update ($localtsval, tsval$) is {
17: **if** $tsval.ts > localtsval.ts$ **then** $localtsval := tsval$}

Figure 3. Code of server s_i

for at least $b + 1$ different servers. Moreover, unless the reader return $c.val$ such that c is $safeFrozen$ (but c is rather $safe$), every other timestamp-value pair c' that is $readLive(c, i)$ for some s_i with a higher timestamp (or the value $v' \neq v$ with the same timestamp) must be deemed $invalidw$ (line 8, Fig. 2) and $invalidpw$ (line 9, Fig. 2), i.e., $highCand(c)$ must hold (lines 10 and 18, Fig. 2). The predicate $invalidw(c)$ holds if $S-t$ servers respond (either in pw_i or w_i variables) with a timestamp-value pair with a timestamp less than $c.ts$ or with the same timestamp as $c.ts$ but with a value different than $c.val$. Similarly, the predicate $invalidpw(c)$ holds if $S-b-t$ servers respond in their pw fields with a timestamp-value pair with a timestamp less than $c.ts$ or with the same timestamp as $c.ts$ but with a value different than $c.val$.

When the reader selects a value $c.val$ that is safe to return, and if this occurs at the end of the first round of the READ invocation, the reader evaluates the predicate $fast(c)$ (defined in line 7, Figure 2), to determine whether it can skip the writeback procedure. The $fast(c)$ holds if c appears in at least $b+1$ server vw fields or $2b+t+1$ servers' pw fields. If the READ rd is *lucky* and at most f_r servers are faulty, the rd is guaranteed to terminate after only one round of the READ invocation, and, hence, rd will be *fast*.

Indeed, if a last complete WRITE that preceded rd was a *fast* WRITE wr (resp., if wr is *slow*), wr has written pw (resp., vw) fields of at least $S - f_w = t + 2b + f_r + 1$ servers (resp., $S - t = t + b + 1$). If a READ is *lucky* and at most f_r servers are faulty, out of these $S - f_w$ servers at least $S - f_w - f_r = t + 2b + 1$ (resp., $S - t - f_r \geq S - 2t = b + 1$) are correct and will send the $READ\_ACK$ message containing pw (resp., vw) (no new values are written after

wr until rd completes, since rd is contention-free) in the first round of READ. Since rd is synchronous, the reader receives all the responses from all correct servers before the expiration of timer. Hence, the predicate $fastpw(pw)$ (resp., $fastvw(vw)$) holds, as well as predicates $safe()$ and $highCand()$ and rd returns $pw.val$ (resp., $vw.val$) without writing it back.

Otherwise, if rd is not *lucky*, or more than f_r servers fail, the reader may have to write back the value to servers. The writeback follows the communication pattern of the WRITE algorithm (lines 26-28, Fig. 2).

4 Upper Bound

Our upper bound $f_w + f_r \leq t - b$ limits the number of actual server failures that *fast lucky* read/write operations can tolerate, in any optimally resilient atomic storage implementation. In short, the principle lying behind this bound is that the system is asynchronous in general and since malicious servers may change their state to an arbitrary one, they can impose on readers a value that was never written, in case the *fast* operations skip too many servers. In the following, we first precisely state our upper bound and then proceed by proving it.

Proposition 2. Let I be any optimally resilient implementation of a SWMR atomic wait-free storage, with the following properties: (1) in any partial run in which at most f_w servers fail, every *lucky* WRITE operation is *fast*, and (2) in any partial run in which at most f_r servers fail, every *lucky* READ operation is *fast*. Then, $f_w + f_r \leq t - b$.

Proof. Let I be any implementation that satisfies properties (1) and (2) of Proposition 2, such that $f_w + f_r > t - b$. We prove the case where $b > 0$ (the case $b = 0$ is similar and is omitted; see full paper [13] for details). Since I uses $2t + b + 1$ servers we can divide the set of servers into five distinct sets: B_1 that contains at least one and at most b servers, B_2 (resp., T_1) that contains at most b (resp., t) servers, and F_r (resp., F_w) that contains exactly $f_r \leq t$[5] (resp., $f_w \leq t$) servers. Without loss of generality, assume that each of these sets contains only one server. If a set has more than one server, we simply modify the runs in a way that all processes inside a set receive the same set of messages, and if they fail, they fail at the same time, in the same way; the proof also holds if any of the sets B_2, T_1, F_r and F_w are empty, as long as $|F_w| + |F_r| > t - b$.

Let r_1 be the run in which all servers are correct except F_w, which fails by crashing at the beginning of the run. Furthermore, let wr_1 be the *lucky* WRITE operation invoked by the correct writer in r_1 to write a value $v_1 \neq \perp$ (where $\perp$ is the initial value of the storage) in the storage and no other operation is invoked in r_1. By our assumption on I, wr_1 completes in tr_1, say at time t_1, and, moreover, wr_1 is *fast*. According to the proposition, wr_1 completes after receiving responses to the first message sent to correct servers (B_1, B_2, T_1 and F_r). Note that the messages that the writer sends to servers during the first round of wr_1 must not contain authenticated data. In r_1, depending on the implementation I correct servers are allowed to exchange arbitrary number of messages after sending the replies to the writer. We denote the set of messages servers exchange among themselves executing some operation op as X_{op}.

Let r'_1 be the partial run that ends at t_1, such that r'_1 is identical to r_1 up to time t_1, except that in r'_1: (1) server F_w does not fail, but, due to asynchrony, all messages exchanged during wr_1 between F_w and the writer remain in transit, and (2) all messages from X_{wr_1} remain in transit. Since the writer cannot distinguish r_1 from r'_1, wr_1 completes in r'_1 at time t_1.

Let r_2 be the partial run that extends partial run r'_1 such that: (1) F_r fails by crashing at t_1, (2) rd_1 is a *lucky* READ operation invoked by the correct reader $reader_1$ after t_1, (3) rd_1 is *fast* and completes at time t_2[6], (4) no additional operation is invoked in r_2, (5) r_2 ends at t_2, (6) all messages that were in transit in r'_1 remain in transit in r_2.

Let r'_2 be the partial run, identical to r_2, except that in r'_2: (1) server F_r does not fail, but, due to asynchrony, all messages exchanged during rd_1 between F_r and the $reader_1$ remain in transit, and (2) all the messages from X_{rd_1} are in transit in r'_2. Since the $reader_1$ and all servers, except F_r, cannot distinguish r_2 from r'_2, rd_1 completes in r'_2 at time t_2 (note that, both in r_2 and r'_2, $reader_1$ and all other servers do not receive any message from F_r).

Let r''_2 be the partial run, identical to r'_2, except that in r''_2: (1) the writer fails during wr_1 and its messages are never delivered to F_r. Since $reader_1$ and all servers, except F_r, cannot distinguish r'_2 from r''_2, rd_1 completes in r'_2 at time t_2 (note that, both in r'_2 and r''_2, $reader_1$ and all other servers do not receive any message from F_r).

Consider now a partial run r_3, slightly different from r''_2, in which the writer (resp., $reader_1$) fails during wr_1 (resp., rd_1) such that the messages sent by the writer (resp., $reader_1$) in wr_1 (resp., rd_1) are delivered only to B_1 (resp., B_1 and F_w) - other servers do not receive any message from the writer (resp., $reader_1$). We refer to the state of B_1 right after the reception of the message from the writer as to σ_1. In r_3, T_1 crashes at the beginning of the partial run. Assume that the writer fails at time t_{fail_w} and that $reader_1$ fails at time $t_{fail_r} > t_{fail_w}$. In r_3, by the time t_{fail_r}, servers B_2 and F_r did not send nor receive any message. Let rd_2 be a READ operation invoked by the correct reader $reader_2$ at

[5] Recall that, in our model, at most t servers can be faulty in any run.

[6] Note that rd_1, according to the proposition, must be *fast*, even if messages that were in transit in r'_1 are not delivered by t_2.

time $t'_3 > max(t_{fail_r}, t_2)$. Since the only faulty server in r_3 is T_1, rd_2 eventually completes, possibly after the messages in X_{wr_1} and X_{rd_1} are delivered. Assume rd_2 completes at time t_3 and returns v_R.

Let r_4 be the partial run, identical to r''_2, except that in r_4: (1) a READ operation rd_2 is invoked by the correct reader $reader_2$ at t'_3 (as in r_3), (2) due to asynchrony all messages sent by T_1 to $reader_2$ and other servers are delayed until after t_3 and (3) at the beginning of r_4, B_2 fails maliciously: B_2 plays according to the protocol with respect to the writer and $reader_1$, but to all other servers and $reader_2$, B_2 plays like it never received any message from the writer or $reader_1$; otherwise, B_2 respects the protocol. Note that $reader_2$ and the servers F_w, F_r and B_1 cannot distinguish r_4 from r_3 and, hence, rd_2 terminates in r_4 at time t_3 (as in r_3) and returns v_R. On the other hand, $reader_1$ cannot distinguish r_4 from r''_2 and, hence, rd_1 is *fast* and returns v_1. By atomicity, as rd_1 precedes rd_2, v_R must equal v_1.

Consider now partial run r_5, identical to r_3, except that in r_5: (1) wr_1 is never invoked, (2) B_1 fails maliciously at the beginning of r_5 and forges its state to σ_1; otherwise, B_1 sends the same messages as in r_3, and (3) T_1 is not faulty in r_5, but, due to asynchrony, all messages sent by T_1 to $reader_2$ and the other servers are delayed until after t_3. The reader $reader_2$ and the servers F_w, F_r and B_2 cannot distinguish r_5 from r_3, so rd_2 completes at time t_3 and returns v_R, i.e., v_1. However, by atomicity, in r_5 rd_2 must return $\perp$. Since $v_1 \neq \perp$, r_5 violates atomicity.

5 Discussion

Our tight bound on the best-case complexity of an atomic storage raises several questions, which we discuss below.

Tolerating malicious readers. While it is pretty obvious that a malicious writer can always corrupt the storage, it is appealing to figure out whether it is feasible to tolerate malicious readers.

The problem is basically the following: consider a complete write followed by a read from a malicious reader that writes back the value to the servers, which itself is followed by a read from a correct reader. Our algorithm does not ensure that the malicious reader will not write back a value that was never written by the writer. Hence, a correct reader might return the value written back by the malicious reader instead of the last written value (thus violating atomicity).

In fact, we are not aware of any optimally resilient atomic implementation that tolerates malicious readers without using data authentication or intercommunication among servers. It would be interesting to devise an algorithm that allows *fast lucky* operations and tolerates malicious readers, in a model where server intercommunication is possible.

Authentication. Our upper bound $f_w + f_r \leq t - b$ of Section 4 is based on the possibility for malicious servers to get to an arbitrary state. If we relax the guarantees of an atomic storage, and accept violations of atomicity with a very small probability, we could benefit from data authentication primitives, such as digital signatures [22] or message authentication codes (MACs). Roughly speaking, this would prevent malicious servers from impersonating the writer (run r_5, Section 4), and hence circumvent our upper bound. However, since the generation of digital signatures is (computationally) expensive, it may impair the benefits of expediting operations in a single communication round (besides, malicious servers would have to be assumed computationally bounded).

On the other hand, (computationally less expensive [7]) MACs might appear to circumvent our upper bound: since clients are non-malicious, they could share a symmetric key (unknown to servers) to prevent malicious servers from forging values (with a very high probability). However, our upper bound requires *every lucky* operation to be *fast* (provided at most f_w/f_r server failures): it is not clear how clients can distribute the secret key while preserving this requirement (recall that any number of clients can fail, and hence clients would have to establish the key through servers). Moreover, MACs are not suitable for solving the malicious readers issue, since in this case, roughly, computational overhead of MACs grows proportionally to the number of readers. In the following, we discuss an alternative approach to boosting thresholds f_w and f_r, without relying on any data authentication.

Trading (few) reads. Our upper bound proof of $f_w + f_r \leq t - b$ heavily relies on the fact that we require *every lucky* operation (in particular, every *lucky* read operation) to be *fast*. In fact, if we allow a certain number, yet just a small fraction, of *lucky* read operations to be *slow*, we can drastically increase the sum of the thresholds f_w and f_r: $f_w \leq t - b$ and $f_r \leq t$. Basically, our very same atomic implementation also ensures that: (1) every *lucky* write can be *fast* given that at most $f_w = t - b$ servers are faulty, and (2) in any sequence of n $(0 < n < \infty)$ consecutive *lucky* reads, there is at most one *slow lucky* read (regardless of the number of server failures, i.e., $f_r = t$).[7]

These bounds ($f_w \leq t - b$ and $f_r \leq t$) are also tight in the following sense. No optimally resilient safe storage algorithm can have every *lucky* write be *fast* despite the failure of $f_w > t - b$ servers.[8]

Trading writes. In our algorithm, *unlucky* writes execute in three communication round-trips. In this sense, our algorithm does not degrade gracefully as it is known that

[7] More details can be found in the full paper [13].

[8] The proof can be found in the full paper [13].

an atomic storage implementation (for $b = t$) can be optimally resilient with two communication-round trips for every write. In the full paper [13], we show that the three communication round-trips worst-case complexity of the write operation is inherent in the general case where $b \neq t$, to optimally resilient algorithms where every *lucky* read is *fast* despite the failure of at least one server.

It is also natural to ask if our upper bound can be circumvented for the *lucky* reads, if we are willing to trade certain, or even all writes. In fact it is easy to modify our algorithm such that writes are *slow* (by removing line 7, Fig. 1) and ensure that *every lucky* read is *fast* (i.e., $f_r = t$).

Regularity vs Atomicity. The problem of malicious readers can easily be solved by weakening the guarantees of the storage implementation. Namely, our algorithm can easily be modified such that: (1) it tolerates malicious readers and (2) the number of actual server failures every *lucky* write (resp., read) operation tolerate f_w (resp., f_r) is $t - b$ (resp., t). This can be achieved by simple modifications of our algorithm, most notably by removing the writeback procedure in the read implementation. The key idea in achieving optimally resilient wait-free implementation while tolerating malicious readers is to allow readers to modify the state of servers without impacting other readers (by using our *freezing* mechanism). The price for these improvements is to trade atomicity for regularity.[9]

Contending with the ghost. If the writer fails without completing a write wr, every subsequent read operation rd is, according to our definition, considered under contention. Therefore, no rd is *lucky* and no rd is guaranteed to be *fast*. Interestingly, in our algorithm, at most three synchronous read operations by some reader r_j, invoked after the failure of the writer, need to be *slow*. Hence, our algorithm quickly overcomes the issues of the writer failure and restores its optimal performance.[10]

6 Related Work

The area of robust shared storage over unreliable components is not new. Original work considered tolerating crash failures of the servers [2]. More recent work considered tolerating arbitrary [16, 19] server failures. We recall here several results about such implementations that are close to ours. We discuss the implementations that do not use data authentication and implement *fast* read/write operations, and we compare those to our algorithm. For an accurate comparison of algorithm performance, unless explicitly stated otherwise, we assume a SWMR setting.

In [21], Martin et al. proved that no safe [17][11] storage implementation is possible if the available number of servers is $S \leq 3t$, considering the case where $b = t$. When $b \neq t$, it is not difficult to extend [21] and show that any safe storage implementation requires at least $2t + b + 1$ servers, establishing an optimal resilience lower bound for any storage implementation in an asynchronous system. Furthermore, Martin et al. presented in [21] a MWMR optimally resilient atomic storage implementation, called SBQ-L, that uses $3t + 1$ servers to tolerate $b = t$ arbitrary server failures without using data authentication, that is not wait-free. In contrast, we present a wait-free optimally resilient implementation that enables *fast* reads and *fast* writes under best-case conditions without using data authentication.

The relationship between resilience and *fast* operations, in the case where $b = t$, was analyzed by Abraham et al. in [1]. They showed that, in order for every write operation to be *fast*, at least $4t + 1$ servers (actually based shared objects) are required even for the case of a single-writer-single-reader (SWSR) safe storage. Furthermore, they used $3t + 1$ passive discs to implement a SWMR wait-free safe and a FW-terminating[12] regular implementations without using data authentication. In their algorithms, writes are never *fast*, but every contention-free and synchronous (i.e., *lucky*) read operation is *fast* despite the actual failure of t shared discs. In contrast, we provide stronger, atomic wait-free, semantics and achieve *fast* synchronous writes, in addition to achieving *fast lucky* reads. To achieve this, besides using some novel techniques (e.g., the "freezing" mechanism as well as the *fast* writing), our algorithm makes use of some techniques established by [1].

In [12], Goodson et al. described an implementation of a wait-free MWMR atomic storage, assuming $2t + 2b + 1$ servers (thus not being optimally resilient). In [12], *lucky* reads are *fast*, in the case there are no server failures $f_r = 0$. In our SWMR setting, this implementation can trivially be modified to allow every write operation to be *fast*. The wait-freedom of the atomic storage of [12] relies on the fact that servers store the entire history of the shared data structure, which is not the case with our algorithm. Moreover, our implementation is optimally resilient and achieves *fast lucky* read/write operations while maximizing the number of server failures that *fast* atomic operations can tolerate in optimally resilient implementations. The algorithm of [12] tolerates *poisonous writes* [21] performed by malicious clients to write inconsistent data into servers, but does not solve the issue of malicious readers that we pointed out in Section 5. The algorithm of [12] uses erasure coding in which servers store data fragments, instead of full repli-

[9] More details can be found in the full paper [13].

[10] More details can be found in the full paper [13].

[11] Roughly, in a safe storage, a read must return the last value written, or any value if it is concurrent with a write.

[12] Roughly, FW-termination requires only that read operations terminate when a finite number of writes are invoked.

cation, to improve bandwidth consumption and storage requirements. Our algorithm can easily be modified to support erasure coding, along the lines of [8, 12].

In [5], Bazzi and Ding presented a MWMR atomic implementation that used $4t + 1$ servers ($b = t$ case), and that can trivially be modified, in the SWMR setting, to achieve *fast* writes whenever there is no concurrency. However, in [5] reads are never *fast* and the implementation is not wait-free.[13] Bazzi and Ding also suggested in [6], an improved, wait-free, version of their algorithm, that still uses $4t + 1$ servers, and ensures *fast* writes when there is no contention; *fast* reads are not considered.

In [11], Dutta et al. considered atomic storage implementations that implement *fast* operations (even in unlucky situations). They derived a tight resilience bound that limits the number of readers that can be supported by such an implementation. Namely, to support R readers, any such emulation must make use of at least $(R + 2)t + (R + 1)b + 1$ servers. The implementation presented in [11] uses data authentication and is clearly not optimally resilient. In this paper, we focus only on optimally resilient implementations that (1) support any number of readers, (2) do not use data authentication and (3) are optimized for *lucky* operations.

7 Acknowledgments

We are very thankful to Partha Dutta and the anonymous reviewers for their very helpful comments.

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

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

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[5] 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.

[13] Both [21] and [5] give wait-free versions of their algorithms assuming data authentication.

[6] R. A. Bazzi and Y. Ding. Brief announcement: Wait-free implementation of multiple-writers/multiple-readers atomic Byzantine data storage systems. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 353–353, New York, NY, USA, 2005. ACM Press.

[7] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. *Lecture Notes in Computer Science*, 1666:216–233, 1999.

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

[9] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

[10] P. Dutta and R. Guerraoui. The inherent price of indulgence. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[11] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolic. 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.

[12] 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.

[13] R. Guerraoui, R. R. Levy, and M. Vukolic. 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.

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

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

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

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

[18] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.

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

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

[21] 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.

[22] 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.