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*

Technical Report LPD-REPORT-2005-005
December 09, 2005
last revision April 20, 2006

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 fail by crashing. We define a *lucky* operation (read or write) as one that runs synchronously and without contention. 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 $f_w + f_r \leq t - b$.

Keywords: Atomic storage, Shared-memory emulations, Arbitrary failures, Optimal resilience, Time-complexity.

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, 10, 19].

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 [3, 20]. 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) [2, 14, 17]. 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 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

[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.

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 [14,15,17,20] 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 kth WRITE in a run ($k \geq 1$), and by val_k the value written by the kth 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. This is followed by detailed descriptions of the WRITE and READ implementations. Finally, in Section 3.4, we prove the correctness of our algorithm.

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 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.

Initialization:
1: $pw := w := \langle ts_0, \perp\rangle$; $ts := ts_0$; $T := timer()$; $frozen := \emptyset$;
2: $\forall r_j | r_j \in readers : read_ts[r_j] := tsr_0$

WRITE(v) **is** {
3: **inc**(ts); $pw := \langle ts, v\rangle$; **trigger**($T$) % *pre-write (PW) phase*
4: send $PW\langle ts, pw, w, frozen\rangle$ to all servers
5: **wait for** $PW_ACK_i\langle ts, newread\rangle$ from $S-t$ servers **and expired**(T)
6: $frozen := \emptyset$; $w := \langle ts, v\rangle$
7: *freezevalues*()
8: **if** $PW_ACK_i\langle ts, *\rangle$ received from $S - f_w$ servers **then return**(OK)

9: **for** $round$= 2 to 3 **do** % *write (W) phase*
10: send $W\langle round, ts, pw\rangle$ to all servers
11: **wait for** reception of $WRITE_ACK_i\langle round, ts\rangle$ from $S-t$ servers
12: **return**(OK) }

freezevalues() **is** {
13: $(\forall r_j, |\{i : (PW_ACK_i.ts = ts) \wedge ((\langle r_j, ts_{r_j}\rangle \in$
$\in PW_ACK_i.newread) \wedge (ts_{r_j} > read_ts[r_j]))\}| \geq b+1$ **do**
14: $read_ts[r_j] := b + 1^{st}$ highest value ts_{r_j}
15: $frozen := frozen \cup \langle r_j, pw, read_ts[r_j]\rangle$ }

Fig. 1. WRITE implementation (writer)

In the PW phase, the writer awaits both for valid responses [3] to the PW message from $S-t$ different servers and the expiration of the timer. The writer completes the PW phase by executing

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

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.

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) 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.

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 $safeFrozen$), 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

```
Definitions and Initialization:
1:  readLive(c, i) ::= (pw_i = c) ∨ (w_i = c)
2:  readFrozen(c, i) ::= (frozen_i.pw = c) ∧ (frozen_i.tsr = tsr)
3:  safe(c) ::= |{i : readLive(c, i)}| ≥ b + 1
4:  safeFrozen(c) ::= |{i : readFrozen(c, i)}| ≥ b + 1
5:  fastpw(c) ::= (|{i : pw_i = c}| ≥ 2b + t + 1)
6:  fastvw(c) ::= (|{i : vw_i = c}| ≥ b + 1)
7:  fast(c) ::= fastpw(c) ∨ fastvw(c)
8:  invalidw(c) ::= |{i : ∃c' : readLive(c', i)∧
     ∧(c'ts < c.ts ∨ (c'.ts = c.ts ∧ c'.v ≠ c.v))}| ≥ S − t
9:  invalidpw(c) ::= |{i : ∃c' : pw[i] = c'∧
     ∧(c'ts < c.ts ∨ (c'.ts = c.ts ∧ c'.v ≠ c.v))}| ≥ S − b − t
10:  highCand(c) ::= ∀c'∀i : (readLive(c', i) ∧ c'.ts ≥ c.ts ∧ c' ≠ c) ⇒
     ⇒ invalidw(c') ∧ invalidpw(c')
11:  tsr := tsr_0; T := timer();

READ() is {
12:  inc(tsr);
13:  rnd := 0; pw_i := w_i := ⟨ts_0, ⊥⟩,rnd_i := 0, 1 ≤ i ≤ S;
14:  repeat
15:    inc(rnd); if rnd = 1 then trigger(T)
16:    send READ⟨tsr, rnd⟩ to all servers
17:    wait for READ_ACK_i⟨tsr, rnd, ∗, ∗, ∗, ∗⟩ 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 ≠ ∅
20:  c_sel := (c.val : c ∈ C) ∧ (¬∃c' ∈ C : c'.ts > c.ts)
21:  if (¬fast(c) or (rnd > 1)) then writeback(c_sel)
22:  return(c_sel.val)

23:  upon receive READ_ACK_i⟨tsr, rnd', pw', w', vw', frozen'_j⟩ 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⟨round, tsr, c⟩ message to all servers
28:    wait for receive WRITE_ACK_i⟨round, tsr⟩ from S − t servers }
```

Fig. 2. READ implementation (reader r_j)

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.

```
Initialization:
1: pw, w, vw := ⟨ts_0, ⊥⟩; newread := ∅; ; ts_{r_j} := tsr_0
2: ∀r_j|r_j ∈ readers : ⟨frozen_{r_j}.pw, frozen_{r_j}.tsr⟩ := ⟨⟨ts_0, ⊥⟩, tsr_0⟩

3: upon receive PW⟨ts, pw', w', frozen⟩ from the writer
4:    update(pw, pw'); update(w, w')
5:    ∀j : ⟨r_j, pw'_j, tsr'_j⟩ ∈ frozen do
6:       if tsr'_j ≥ ts_{r_j} then ⟨frozen_{r_j}.pw, frozen_{r_j}.tsr⟩ := ⟨pw'_j, tsr'_j⟩
7:    newread := ⋃⟨r_j, ts_{r_j}⟩, for all r_j such that ts_{r_j} > frozen_{r_j}.tsr
8:    send PW_ACK_i⟨ts, newread⟩ to the writer

9: upon receive READ⟨tsr', rnd'⟩ from reader r_j
10:    if (tsr' > ts_{r_j}) and (rnd' > 1) then ts_{r_j} := tsr
11:    send READ_ACK_i⟨tsr', rnd', pw, w, vw, frozen_{r_j}⟩ to r_j

12: upon receive W⟨round, ts, c⟩ 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⟨round, ts⟩ to clnt


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

Fig. 3. Code of server s_i

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).

3.4 Correctness

We first prove atomicity and then we proceed to wait-freedom and complexity. Note that in order for rd to return a value $c.val$, it is necessary that $c \in C$ (lines 18-20, Fig. 2). Therefore, if some complete READ returns $c.val$ then c must satisfy one of the following predicates (line 18, Fig 2): (1) $safe(c) \wedge highCand(c)$, or (2) $safeFrozen(c)$. For simplicity of presentation, in the following, if c satisfies predicate (1), we simply say that *c is live*, and if c satisfies predicate (2), we say that *c is frozen*.

Furthermore, for simplicity of presentation, when we say that a server *responds* to the client we assume that this response is a $PW-ACK$, $WRITE-ACK$ or $READ-ACK$ message. A *valid response* to a writer's $PW\langle ts,*,*,*\rangle$ message is a $PW-ACK$ message with the same timestamp ts, i.e., a $PW-ACK\langle ts,*\rangle$ message. Similarly, a valid response to a $W\langle round,ts,*\rangle$ message or a $WB\langle round,ts,*\rangle$ is a $WRITE-ACK\langle round,ts\rangle$ message. A valid response to a $READ$ message is defined analogously.

Lemma 1. ***No-creation.*** *If a READ rd completes and returns some value v, then either v was written by some WRITE or v is the initial value $\perp$.*

Proof. Suppose by contradiction that some READ returns a value that is neither $\perp$, nor written by some WRITE. In that case, let rd be the first READ (according to the global clock) to select such a value v at line 20, Fig. 2 at time t_v (it is not difficult to see that such a rd exists). Therefore, if any READ operation has written back a value up to t_v, this value must have been $\perp$, or it has been written by some WRITE. Without loss of generality, assume that v is returned as $v = c.val$, where $c.ts = ts$. Then, according to line 18, Fig. 2, for the timestamp value pair c, either $safe(c)$ or $safeFrozen(c)$ hold, i.e., either $b+1$ servers have sent a $READ-ACK$ message containing

c in either pw or w ($safe(c)$) fields, or $frozen$ ($safeFrozen(c)$) field, including at least one non-malicious server s_i. Note that non-malicious servers update their pw, w and $frozen$ fields only when they receive a PW or W message from the writer, or a writeback message (WB) from the reader. Since until time t_v no reader has sent any writeback message containing a value that is neither $\perp$ nor written by some WRITE, we conclude that c was written by some writer or it was never updated by s_i, so it is $\perp$. A contradiction. □

Lemma 2. ***No ambiguity.*** *No two non-malicious servers ever store different values with the same timestamp.*

Proof. Note that the writer never assigns different values to the same timestamp (lines 4-5, Fig. 1). By Lemma 1, when (if) writing back a value, the readers always write back a value along with the timestamp the writer assigned to it. Therefore, it is impossible that two non-malicious servers store different values with the same timestamp in their pw, w, vw or $frozen$ variables. □

Lemma 3. ***Non-decreasing timestamps.*** *Non-malicious servers never replace a newer value with an older one in their pw, w or vw fields.*

Proof. Obvious from server code inspection, Figure 3. □

Lemma 4. ***Frozen is concurrent.*** *If a complete READ rd returns c.val, such that c is frozen, then c is written by some WRITE wr concurrent with rd.*

Proof. We prove this lemma by contradiction. Suppose wr is not concurrent with rd. There are two possibilities: (1) wr precedes rd, and (2) rd precedes wr. Without loss of generality we consider the case where the reader r_j executes rd.

Consider case (1). Note that every READ by r_j has its distinct, monotonically increasing timestamp. Let the timestamp for rd be tsr_{rd}. A non-malicious server changes its ts_{r_j} variable only if it receives the $READ$ message from r_j containing a timestamp greater than ts_{r_j} (line 10, Fig. 3). Therefore, a non-malicious server sets ts_{r_j} tp tsr_{rd} only once rd started. In addition, the writer "freezes" the value for tsr_{rd}, and then sends it within some later WRITE wr to servers, only when it receives at least $b+1$ server $PW-ACK$ messages containing the pair $\langle r_j, tsr'' \rangle$ in the $newread$ fields, where $tsr'' \geq tsr_{rd}$ and the $b+1$st highest value is tsr_{rd}. Among these $b+1$ tsr'' values, at least one is from a non-malicious server. Therefore, rd must have already started when the writer freezed the value for tsr_{rd}. This is a contradiction with the assumption that wr precedes rd.

Consider now case (2). Note that the writer sends the $\langle r_j, c, read{-}ts[r_j] \rangle$ (lines 6 and 21, Fig. 1) in a $frozen$ field of the PW message of the WRITE with a timestamp $ts' = c.ts + 1$. Note also that rd by r_j returns $c.val$ such that c is frozen only if the latest copy of the variable $frozen_{r_j}$ of at least $b+1$ servers contains $frozen_{r_j}.pw = c$ and $frozen_{r_j}.tsr = tsr_{rd}$, where tsr_{rd} is a timestamp of rd. This includes at least one non-malicious server s_i that updates its $frozen_{r_j}$ variable only upon receiving the PW message for a WRITE with a timestamp ts'. Therefore, rd has not yet completed when a WRITE with a timestamp ts' was invoked. Since wr that wrote $c.val$ precedes a WRITE with a timestamp ts', we conclude that rd cannot precede wr. A contradiction. □

Lemma 5. ***Locking a pw value.*** *If a set X of at least $t+b+1$ non-malicious servers set their local variable pw such that $pw.ts \geq pw'.ts$ at time t', then a complete READ rd invoked after t' cannot return a value $pw''.val$ such that $pw''.ts < pw'.ts$.*

Proof. First, consider the case where pw'' is *live*. Every server in X stores pw such that $pw.ts \geq pw'.ts$ before rd is invoked. In addition, the timestamps of pw variables at non-malicious servers are non-decreasing (Lemma 3). The response of each server in X to rd, if any, contains pw, such

that $pw.ts \geq pw'.ts$. Since a reader in every round of READ awaits responses from $S-t$ servers (line 17, Fig. 2), at least $b+1 \geq 1$ servers from the set X will respond to every round of rd. Let c be the smallest timestamp-value pair contained in the pw field of some response to rd from a non-malicious server s_i, for which $c.ts \geq pw'.ts$. We prove, by contradiction, that c is not *invalidpw*. By definition of *invalidpw*, a set Y of at least $S-b-t$ servers must have responded with values c' in their pw fields, such that $c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Since X contains only non-malicious servers and $|X| \geq t+b+1$, and as $|X|+|Y| > S$, X and Y intersect in at least 1 (non-malicious) servers, so at least one of the responses from Y is from the non-malicious server s_j that belongs to the set X. Therefore, $pw_j.ts \geq c.ts$ and, thus, $pw_j.ts = c.ts \wedge pw_j.val \neq c.val$. However, as both s_i and s_j are non-malicious, this is impossible (Lemma 2). A contradiction.

Therefore, c where $c.ts \geq pw'.ts$ is not *invalidpw* and READ rd can not return any pw'' such that: (1) pw'' is live, and (2) $pw''.ts < pw'.ts$.

Now suppose that pw'' is *frozen*. Let $pw_1.ts$ be the lowest timestamp of every $pw.ts\ gepw'.ts$ in X. Since non-malicious servers store only values written by some WRITE, we conclude that a WRITE with a timestamp $pw_1.ts$ was invoked before rd. If rd returns pw'' such that $pw''.ts < pw'.ts \leq pw_1.ts$, then a WRITE with a timestamp $pw''.ts$ precedes rd. This violates Lemma 4. A contradiction. □

Lemma 6. ***Locking a w value.*** *If a set of X of at least $t+1$ non-malicious servers have set their local variables pw and w such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$ at time t', a complete READ rd invoked after t cannot return a value $w''.val$ such that $w''.ts < w'.ts$.*

Proof. First consider the case where pw'' is *live*. Every server in X stores pw and w such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$ before rd is invoked. In addition, the timestamps of variables pw and w at non-malicious servers are non-decreasing (Lemma 3), each server in X, if it responds to rd, its response in every round will contain pw and w, such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$. Since a reader in every round of READ awaits responses from $S-t$ servers (line 17, Fig. 2), at least 1 server from the set X will respond to every round of rd. Let c be the smallest timestamp-value pair returned in the pw or w field by a non-malicious server s_i, for which $c.ts \geq w'.ts$. We prove, by contradiction, that c is not *invalidw*. By definition of *invalidw*, a set Y of at least $S-t$ servers must have responded with values c' in their pw or w fields, such that $c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Since X contains only non-malicious servers and $|X| \geq t+1$, $|X|+|Y| > S$, so at least one of the responses from Y is from the non-malicious server s_j that belongs to the set X. Therefore, $pw_j.ts \geq c.ts$ and $w_j.ts \geq c.ts$ and, thus, $pw_j.ts = c.ts \wedge pw_j.val \neq c.val$ or $w_j.ts = c.ts \wedge w_j.val \neq c.val$. However, as both s_i and s_j are non-malicious, this is impossible (Lemma 2). A contradiction.

Therefore, c where $c.ts \geq w'.ts$ is not *invalidpw* and READ rd cannot return any w'' such that: (1) w'' is live, and (2) $w''.ts < w'.ts$.

Now suppose that pw'' is *frozen*. Let $w_1.ts$ be the lowest timestamp of every $w.ts\ gew'.ts$ in X. Since non-malicious servers only store values written by some WRITE, we conclude that a WRITE with a timestamp $w_1.ts$ was invoked before rd. If rd returns w'' such that $w''.ts < w'.ts \leq w_1.ts$, then a WRITE with a timestamp $w''.ts$ precedes rd. This violates Lemma 4. A contradiction. □

Lemma 7. ***Atomicity of READ with respect to WRITEs.*** *If a READ rd is complete and it succeeds some complete $wr = WRITE(v)$, then rd does not return a value older than v.*

Proof. First note that the timestamps monotonically increase at the writer and all the values that any reader returns (and, hence, all the values that it writes back) are written by the writer. Therefore, timestamps associated to the values by the writer in line 4, Fig 1, order the values that readers return. Now we show that rd does not return $c.val$, such that $c.ts < ts$, where ts is associated to v in line 4, Fig 1 of $WRITE(v) = wr$. We consider two cases: (1) c is *live* and (2) c is *frozen*.

1. $safe(c) \wedge highCand(c)$ (c is live). First, suppose that wr is a *fast* WRITE. Then, $\langle ts, v\rangle$ is written to pw fields of at least $S - f_w$ servers, out of which a set X, of size at least $S - f_w - b$ are non-malicious. Since $f_w \leq t - b$, we obtain $S - f_w - b \geq S - t = t + b + 1$, and we can apply Lemma 5 ($pw'.ts$ being ts), and conclude that rd does not return a value with a timestamp less than ts, i.e., older than v.
   Now, let wr be a WRITE that completed in two phases. Then, $\langle ts, v\rangle$ is written to pw, w and vw fields of at least $S-t = t+b+1$ servers, out of which a set X, of size at least $S-t-b = t+1$, contains only non-malicious servers. We can apply Lemma 6 ($w'.ts$ being ts), and conclude that rd does not return a value with a timestamp less than ts, i.e., older than v. Therefore, no c' such that $c'.ts < ts$ can satisfy the $highCand$ predicate, and no such c can be returned as live. A contradiction.
2. $safeFrozen(c)$ (c is frozen). Due to lemma 4, c has been written by some WRITE wr' concurrent with rd. Since (1) the writer assigns non-decreasing timestamps to values, (2) wr precedes rd and (3) rd is concurrent with wr', we conclude that wr precedes wr' and $c.ts > ts$. □

Lemma 8. ***READ hierarchy.*** *If a READ rd is complete and it succeeds some complete READ rd' that returns v', then rd does not return a value older than v'.*

Proof. Note that a complete READ rd does not write back a value $c.val$ it returns if rd is *fast*, i.e., if rd selects c at line 20, Fig. 2 at the end of first round ($rnd = 1$), and if $fast(c)$ holds (i.e., if either $fastpw(c)$ or $fastvw(c)$ hold). We now show that rd does not return $c.val$, such that $c.ts < ts'$, where ts' is associated to v' in line 4, Fig. 1 of the WRITE that wrote v'. We consider two cases: the first where rd' is *fast* and the second in which rd' is *slow* and writesback $c' = \langle ts', v'\rangle$ (lines 26-28, Fig. 2).

1. rd' is *fast*. First we consider the case where $fastpw(c')$ holds and then the one in which $fastvw(c')$ holds in rd'.
   a. $fastpw(c')$ holds in rd'. In this case, in rd' a reader has received at least $t+2b+1$ different server responses that contain c' in their pw fields. Out of these, a set X of at least $t + b + 1$ servers are non-malicious. Applying Lemma 5, we conclude that rd returns a value $c.val$ such that $c.ts \geq c'.ts$.

   b. $fastvw(c')$ holds in rd'. In this case, in rd' a reader has received at least $b + 1$ different server responses that contain c' in their w fields. This includes at least one non-malicious server s_i. According to the server code, non-malicious servers modify their vw fields to c' only when they receive W or WB message with a $round = 3$ (line 15, Fig. 3) from the writer or the reader. In any case, a set X of at least $t + 1$ non-malicious servers have already all set their pw and w variables to c' when receiving a W or WB message with $round = 2$. Hence, we can apply Lemma 6, and conclude that rd does not return a value with a timestamp less than $c'.ts$, i.e., older than $c'.val$.
2. rd' writesback the value. In this case, in rd' the reader has completed writing back the value $c'.val$ and a set X of at least $t + 1$ are non-malicious have all set their pw, w and vw variables to c' at latest during rd', i.e., before rd is invoked. Hence, we can apply Lemma 6, and conclude that rd does not return a value with a timestamp less than $c'.ts$, i.e., older than $c'.val$. □

Theorem 1. ***Atomicity.*** *The algorithm in Figures 1, 2 and 3 is atomic.*

Proof. By Lemmas 1, 7 and 8. □

We proceed by proving the wait-freedom property.

Theorem 2. ***(Wait-freedom.)*** *The algorithm in Figures 1, 2 and 3 is wait-free.*

Proof. The argument for the wait-freedom of a WRITE operation is based on the assumption that there are at most t faulty servers. In every round of a WRITE, the writer waits for at most $S-t$ valid server responses that the writer is guaranteed to receive eventually. The timer that the writer awaits in line 8, Figure 1, eventually expires and the operation eventually completes.

The argument for the wait-freedom of a READ operation is slightly more involved. We prove that in every run in which at most t servers fail (out of which at most b are malicious), every READ operation rd invoked by the correct reader completes.

We distinguish two cases: (a) the case where the writer issues a finite number of WRITE operations in the run, and (b) the case where there is an infinite (unbounded) number of WRITE operations in the run.

Case (a) - Finite number of WRITES. In this case, since no READ returns (nor writes back) any value not written by some READ there is a WRITE operation with the highest timestamp. Let wr denote the last complete WRITE operation that writes v with timestamp ts (or $v = v_0$, $ts = \perp$ if there is none. We denote with wr' a possible later (incomplete) WRITE that writes v' with ts'.

Assume, by contradiction, that READ rd never returns a value. Then, rd invokes rounds on all correct servers, sending $\langle READ \rangle$ messages infinitely many times. We distinguish two cases: (a.1) value v' is never pre-written (written in the pw fields) into more than b correct servers and (a.2) there is a time t at which wr' is pre-written into $b+1$ or more correct servers.

In case (a.1) let t be the time at which last correct server changed its pw to $\langle ts', v' \rangle$ (by wr' or an incomplete writeback that sends a WB message with a same timestamp-value pair). In both cases, let $t' > t$ be the time at which rd received at least one response from every correct server sent after t.

Consider case (a.1). First we prove that $\langle ts, v \rangle$ is $safe$. There are two cases: (a.1.i) first in which wr is *fast*, and (a.1.ii) second, in which wr is *slow*.

(i) Consider the case where wr is a *fast* WRITE. Since $\langle ts, v \rangle$ was written to pw fields of at least $S - f_w$ servers, out of which at least $S - f_w - t \geq 2b+1$ are correct. Since $c' = \langle ts', w' \rangle$ is pre-written in the pw field of at most b of these correct servers, from time t' onward, $\langle ts, v \rangle$ appears at least $b+1$ times in $pw[*]$ or $w[*]$ and is safe.

(ii) Now, assume that wr is not a *fast* WRITE. Since $\langle ts, v \rangle$ was written to vw and w fields of at least $S-t$ servers, out of which at least $S - 2t \geq b+1$ are correct. Out of these, no vw or w field ever changes its value (as this would require that c' is pre-written in at least $b+1$ correct servers). Hence, from time t' onward, $\langle ts, v \rangle$ appears at least $b+1$ times in $w[*]$ and is safe.

Moreover, there are at least $S-t$ responses in $w[*]$, from correct servers, with either $\langle ts, v \rangle$ or with a timestamp smaller than ts. Therefore, every timestamp-value pair c, such that $c.ts > ts \vee (c.ts = ts \wedge c.val \neq v)$ is $invalidw$. Finally, there are at least $S-t-b$ responses from correct servers in $pw[*]$ with either $\langle ts, v \rangle$ or with a timestamp smaller than ts. Therefore, every timestamp-value pair c, such that $c.ts > ts \vee (c.ts = ts \wedge c.val \neq v)$ is $invalidpw$. Thus, at the next iteration, $highCand(\langle ts, v \rangle)$ and $safe(\langle ts, v \rangle)$ hold, $\langle ts, v \rangle \in C$ and rd returns, a contradiction.

Now consider case (a.2), where v' is pre-written (by a WRITE or some READ) into $b+1$ or more correct servers. Then, after t', $\langle ts', v' \rangle$ appears at least $b+1$ times in $pw[*]$ and is safe. It is not difficult to see, as no subsequent valid value is present in the system, such that for every pair c'' such that $c''.ts > c'.ts \vee (c''.ts = c'.ts \wedge c''.val \neq c'.val)$ $invalidw(c'')$ and $invalidpw(c'')$ holds. Thus in the next iteration, $\langle ts', v' \rangle \in C$ and READ returns: a contradiction.

Case (b) - Infinite number of WRITES. Suppose, by contradiction, that rd never completes. Let tsr_j be the timestamp of rd (at the correct reader r_j). Note that the reader r_j never invokes a READ rd' with a timestamp $tsr'_j > tsr_j$, and therefore, the writer cannot freeze a value (for the

reader r_j) for a READ with a higher timestamp than tsr_j). Since rd invokes an infinite number of rounds (rounds do not block as in every round the reader awaits $S - t$ responses), eventually, the reader will write tsr_j to every correct server. Since there are $S - t = t + b + 1$ correct servers, in the next WRITE round (as there is an infinite number of WRITES, the writer executes an infinite number of WRITE rounds), the writer reads tsr_j from at least $b + 1$ of these servers, and will freeze the current pw value (pw') for the reader r_j (it is not difficult to see that, for a particular tsr_j, the writer freezes at most one value). Since the channels are reliable and the servers take into account $frozen$ fields of the old PW messages of the writer, eventually, every correct server stores $frozen_{r_j} = \langle pw', tsr_j \rangle$. Since there are $S - t = t + b + 1$ correct servers, in the next READ iteration, r_j reads the same $frozen_{r_j} = \langle pw', tsr_j \rangle$ from at least $b + 1$ correct servers, pw' becomes $safeFrozen$ and rd terminates. A contradiction. □

Now we prove that our algorithm implements *fast* WRITES and *fast* READS. If at most f_w servers are faulty, every synchronous WRITE is *fast*. Moreover, if at most f_r servers are faulty, every *lucky* READ (i.e., the READ that is synchronous and contention-free) is *fast*.

Theorem 3. ***(Fast WRITES.)*** *In the algorithm of Figures 1, 2 and 3, if a complete WRITE is synchronous and at most f_w servers fail by the completion of wr, then wr is* fast*.*

Proof. As at most f_w servers are faulty and the WRITE is synchronous, the writer receives the response from at least $S - f_w$ servers in the PW round before the timer $Timeout$ expires (line 8, Fig. 1) and the WRITE completes in a single round-trip, before the second, W phase. □

Theorem 4. ***(Fast READS.)*** *In the algorithm of Figures 1, 2 and 3, if a complete READ operation rd is synchronous and contention-free (i.e., if rd is* lucky*) and at most f_r servers fail by the completion of rd, then rd is* fast*.*

Proof. First, suppose that the last (complete) WRITE (wr) that precedes rd writes a timestamp-value pair c_{pw}. As rd is contention free, no other WRITE is invoked before the completion of rd, and, hence, no other value stored by a correct server has a timestamp higher than $c_{pw}.ts$ during the execution of rd.

First, suppose that wr is a *fast* WRITE. Thus, c_{pw} was written in the pw field of at least $S - f_w = 2b + t + f_r + 1$ servers out of which at least $S - f_w - f_r = 2b + t + 1$ are correct and respond with a valid $READ - ACK$ message in the first round of rd. Therefore, at the end of round 1, $safe(c_{pw})$ and $fastpw(c_{pw})$ (line 5, Fig. 2) hold. Let c' be any timestamp-value pair such that $\exists i, readLive(c', i), c'.ts \geq c_{pw}.ts \vee (c'.ts = c_{pw}.ts \wedge c'.val \neq c_{pw}.val)$. Then, as there are no WRITES after wr that are concurrent with rd, and as the system is synchronous, all $S - f_r \geq S - t$ correct servers respond before the expiration of the timer (and, thus, round 1) with timestamps less than $c'.ts$ or with a timestamp $c'.val = c_{pw}.ts$ and a value $c_{pw}.val \neq c'.val$. Therefore, at the end of round 1, for every such timestamp-value pair c', the predicates $invalidw(c')$ and $invalidpw(c')$ hold. Finally, predicate $highCand(c_{pw})$ is true at the end of round 1 and predicate $fastpw(c)$ holds and rd returns $c_{pw}.val$.

Now suppose that wr is a two-phase WRITE. Hence, c_{pw} was written in the pw, w and vw fields of at least $S - t = b + t + 1$ servers out of which at least $S - t - f_r \geq S - 2t = b + 1$ are correct and respond with a valid $READ - ACK$ message in the first round of rd. Therefore, at the end of round 1, $safe(c_{pw})$ and $fastvw(c_{pw})$ (line 6, Fig. 2) hold. Let c' be any timestamp-value pair such that $\exists i, readLive(c', i), c'.ts \geq c_{pw}.ts \vee (c'.ts = c_{pw}.ts \wedge c'.val \neq c_{pw}.val)$. Then, as there are no WRITES after wr that are concurrent with rd, and as the system is synchronous, all $S - f_r \geq S - t$ correct servers respond before the expiration of the timer (and, thus, round 1) with timestamps less than $c'.ts$ or with a timestamp $c'.val = c_{pw}.ts$ and a value $c_{pw}.val \neq c'.val$. Therefore, at the end of round 1, for every such a timestamp-value pair c', the predicates $invalidw(c')$ and $invalidpw(c')$

hold. Finally, predicate $highCand(c_{pw})$ is true at the end of round 1 and predicate $fastvw(c_{pw})$ holds and rd returns $c_{pw}.val$. □

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 $f_w + f_r > t - b$. First, we prove the case where $b > 0$. 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 $f_r \leq t$[4] (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 $|F_w| + |F_r| > t - b$.

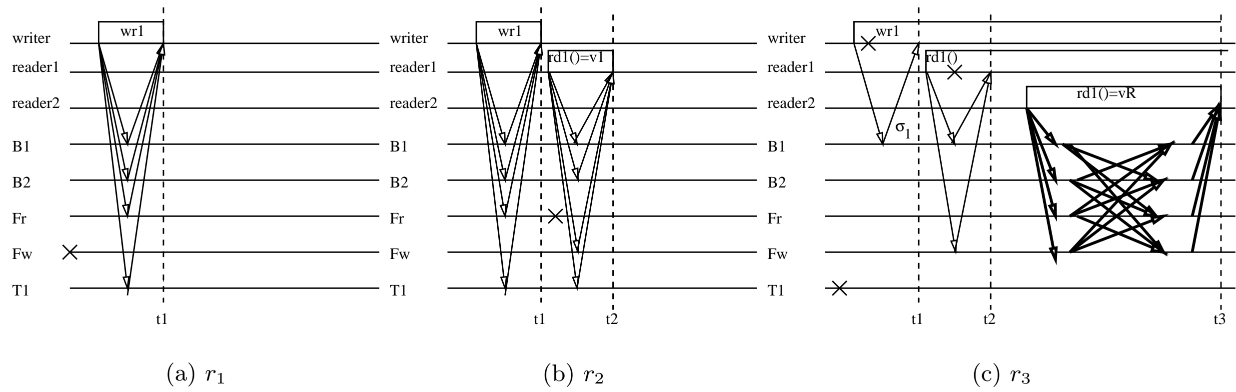
The key runs that we use in our proof are sketched in Figure 4, to help follow the proof. Note that, depending on the implementation I, correct servers are allowed to exchange arbitrary number of messages after sending the replies to the first round message of the *fast* operation to a client. For clarity, these messages are not depicted in Figure 4. Bolded arrows in Figure 4 depict multiple messages exchanged between processes during the particular (*slow*) operation.

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}.

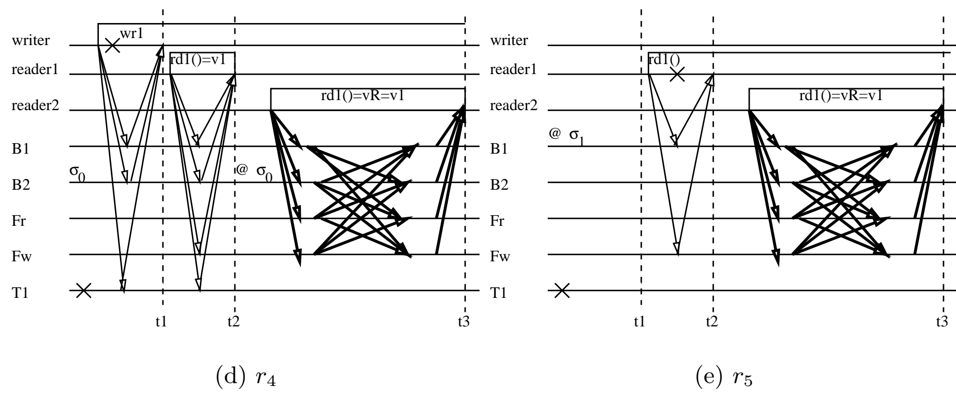
Let r'_1 be the partial run that ends at t_1, such that r'_1 is identical to r_1 up to time t_1, except that in r'_1: (1) server F_w does not fail, but, due to asynchrony, all messages exchanged during wr_1 between F_w and the writer remain in transit, and (2) all messages from X_{wr_1} remain in transit.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

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



(a) r_1 (b) r_2 (c) r_3



(d) r_4 (e) r_5

- single message

- several messages delivered during the operation

@ - server is malicious

× - server/client crashes

(f) Legend

Fig. 4. Illustration of the upper bound proof of Proposition 2

and completes at time t_2[5], (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 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.

Consider now the case where $b = 0$. We can divide the set of servers into three distinct sets: T_1 that contains at most t servers and F_r (resp., F_w) that contains exactly f_r (resp., f_w servers). Consider the partial runs r_{1a} to r_{5a} obtained by modifying the (partial) runs r_1 to r_5, respectively, in such a way that all steps by the servers B_1 and B_2 are erased. It is not difficult to see, that $reader_2$ cannot distinguish r_{5a} from r_{4a} and r_{3a}. By atomicity property, in r_{4a} $reader_2$ must return v_1 as the preceding *fast* READ rd_1 has returned v_1. Therefore, in r_{5a} rd_2 returns v_1 a value that was never written, violating atomicity. □

[5] 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.

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 $f_r \leq t$. First we define the notions of *sequence* of *consecutive lucky* operations.

Definition 1. ***Sequence.*** *Consider n* lucky *READ operations $rd_i, 1 \leq i \leq n$. If, for every i such that $2 \leq i \leq n$, rd_{i-1} precedes rd_i, we say that the ordered set $\{rd_1, \ldots rd_n\}$ is a* sequence *of* lucky *READ operations of length n.*

Definition 2. ***Consecutive.*** *Consider the sequence $S_n = \{rd_1, \ldots rd_n\}$ of* lucky *READ operations of length n. If no WRITE operation is invoked in the time period $[t_{rd_{1inv}}, t_{rd_{nresp}}]$, where rd_1 is invoked at $t_{rd_{1inv}}$ and rd_n completes at $t_{rd_{nresp}}$, we say that S_n is a sequence of* consecutive *lucky READ operations of length n.*

Basically, our very same atomic implementation of Section 3 ensures that: (1) every *lucky* write is *fast* given that at most $f_w = t - b$ servers are faulty, and (2) in any sequence of n ($0 < n < \propto$) consecutive *lucky* reads, there is at most one *slow lucky* read (regardless of the number of server failures, i.e., $f_r = t$). More details can be found in Appendix A.

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. The proof can be found in Appendix B.

Trading writes. 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 8, Fig. 1) and ensure that *every lucky* read is *fast* (i.e., $f_r = t$).

An inherent price of lucky *reads.* Recall that, in our algorithm, servers send messages only in response to clients' messages, i.e., servers do not exchange messages with other servers, nor they send unsolicited messages. In such *data-centric* model, the notion of the number of communication-rounds per operation effectively captures the latency of distributed operations [1, 13].

In our algorithm, *unlucky* writes execute in three communication round-trips. In this sense, our algorithm does not degrade gracefully, since an atomic storage implementation can be optimally resilient with two communication-round trips for every write. However, in Appendix C, we show that the three communication round-trips worst-case complexity of the write operation is in fact inherent to data-centric, optimally resilient algorithms that do not use authentication to tolerate malicious server failures ($b > 0$), and that enable every *lucky* read to be *fast* despite the failure of at least one server (which is precisely the case with our algorithm, if $f_r > 0$).

More specifically, in Appendix C, we show that for any data-centric algorithm that enables every *lucky* read to be *fast* despite the failure of f_r servers, a write operation can be implemented in (at most) two communication-rounds if and only if the total number of servers S is at least $S \geq 2t + b + min(b, f_r) + 1$. Consequently, since our algorithm is optimally resilient (i.e., it makes use of exactly $S_{opt} = 2t + b + 1$ servers), and allows *fast lucky* reads despite the failure of $f_r > 0$ servers (in the general case where $b \neq t$, and $f_w \neq t - b$, since f_r is bounded by $t - b - f_w$), our algorithm cannot have the worst-case complexity of write operation of only two communication round-trips. Therefore, the third communication round-trip is inherent for the write operation in our case.

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 that 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. We show hot to modify our atomic implementation to obtain a regular one in Appendix D.

Multiple writers. It is not clear whether it is possible to have a multi-writer multi-reader (MWMR) implementation where certain write operations may be *fast* (i.e., complete in a single round-trip without using data authentication) in some runs. To the best of our knowledge, none of the existing MWMR implementations implements *fast* write operations under any conditions.

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. More details can be found in Appendix E.

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, 18] 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][6] 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[7] 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 replication, to improve bandwidth consumption and storage requirements. Our algorithm can easily be modified to support erasure coding, along the lines of [8, 12].

[6] Roughly, in a safe storage, a read must return the last value written, or any value if it is concurrent with a write.
[7] Roughly, FW-termination requires only that read operations terminate when a finite number of writes are invoked.

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.[8] 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. Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia 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.
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. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John 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. Rida Bazzi and Yin 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.
6. Rida A. Bazzi and Yin 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. Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. Technical Report RZ 3575, IBM Research, February 2005.
9. Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.
10. Partha Dutta and Rachid 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. Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko 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. Rachid Guerraoui and Marko Vukolic. How fast can a very robust read be? In *To appear in Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC 2006)*. ACM Press, 2006.
14. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

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. Prasad Jayanti, Tushar Deepak Chandra, and Sam 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, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
19. Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.
20. N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
21. Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325. Springer-Verlag, 2002.
22. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

A Trading (few) READS

In this Appendix, we prove that our atomic storage implementation of Section 3 satisfies the following proposition:

Proposition 3. There is an optimally resilient implementation I of a SWMR robust atomic storage, such that:

- in any partial run r in which at most $t-b$ servers fail, every *lucky* WRITE operation is *fast*,
- in any partial run r, for any $n, 0 < n < \propto$, in any sequence S_n of consecutive *lucky* READ operations of length n, there is at most one *slow lucky* READ operation $rd' \in S_n$.

Note that, in Proposition 3, the sum $f_w + f_r$ must no longer equal $t-b$ (as is the case in Proposition 1, Section 3), but rather: (1) f_w equals $t-b$, and (2) implicitly, $f_r = t$.

It can be seen that, when we substitute $f_w = t-b$ and $f_r = t$, all lemmas and theorems of the correctness proof of Section 3.4 still hold, except Theorem 12 (Fast READS).

Therefore, it is left to prove the following theorem.

Theorem 5. ***(Case $f_r = t$: almost all lucky READS are fast.)*** *In the algorithm of Figures 1, 2 and 3, in any partial run r, for any $n, 0 < n < \propto$, in any sequence S_n of consecutive lucky READ operations of length n, there is at most one* slow lucky *READ operation $rd' \in S_n$.*

Proof. Suppose, by contradiction, that there is a partial run r' and a sequence S_k of consecutive *lucky* READ operations of length k ($k \geq 2$), such that there are two *lucky* READ operations $rd', rd'' \in S_k$, such that both rd' and rd'' are *slow*. Without loss of generality, assume rd' precedes rd'' in r'.

As rd' and rd'' are *lucky*, there is no WRITE operation concurrent with either rd' or rd''. Let v be the value written by the last complete WRITE wr that precedes rd' (or $v = \perp$ if there is no such a WRITE) and let ts be the timestamp associated to v by the writer in line 4, Figure 1 (or $ts = ts_0$, if $v = \perp$). By atomicity, rd' returns v. By our assumption on rd', rd' is *slow* and, therefore, rd' writesback $\langle v, ts \rangle$ before it completes. Since rd'' succeeds rd', by the time rd'' is invoked, at least $b+1$ correct servers set their pw, w and vw variables to $\langle v, ts \rangle$. Therefore, at the end of the first round of rd'', $safe(\langle v, ts \rangle)$ and $fastvw(\langle v, ts \rangle)$ hold. Moreover, as no WRITE operation is invoked after wr before rd'' completes (rd' and rd'' belong to the sequence of consecutive *lucky* READ operations, and wr is the last WRITE that precedes rd', that, in turn, precedes rd''), no correct server ever stores in its pw, w or vw fields a value with a higher timestamp than ts. Since rd'' is *lucky*, all correct servers respond in the first round of rd''. Therefore, it is not difficult to see that, for any c' such that $\exists i, readLive(c', i), c'.ts \geq ts \vee (c'.ts = ts \wedge c'.val \neq v)$, predicates $invalidw(c')$ and $invalidpw(c')$ also hold. Therefore, at the end of the first round of rd'' $highCand(\langle v, ts \rangle)$ holds and rd'' returns v at the end of the first round, i.e., rd'' is *fast*. A contradiction. □

A.1 Remarks

Note that our algorithm minimizes the number of *lucky* READ operations that may be *slow* when $f_w = t-b$, $f_r = t$, in the sense that in any sequence of consecutive READ operations at most one *lucky* READ operation may be *slow*. Indeed, if we would require an algorithm such that in any sequence of consecutive READ operations all READS operations be *fast*, such an algorithm would be subject to our upper bound result $f_w + f_r \leq t-b$ of Section 4.

Finally, it can be shown that in our algorithm, once more than $t-b$ servers fail in a partial run r, at time T, and at least one WRITE wr invoked after T completes, every *lucky* READ operation that succeeds wr is *fast* (despite the possible failure of $f_r = t$ servers). This is the case because no WRITE invoked after T will be *fast*. Roughly, the price of a single *slow* READ per sequence

of consecutive *lucky* READS (in case $f_r = t$) is paid in order to have *fast* WRITES (in case that at most $t - b$ servers fail). In a sense, this single *slow* READ in a sequence of consecutive *lucky* READS can be seen as the one that "finishes" the *fast* WRITE.

B Upper Bound on Fast Writes

In this Appendix, we show that no optimally resilient *safe* storage implementation can have *every lucky* WRITE operation be *fast* despite the failure of more than $t - b$ servers.

Proposition 4. If I is an optimally resilient implementation of a SWSR robust safe storage such that: in every partial run r in which at most f_w servers fail, every *lucky* WRITE operation wr is *fast*, then $f_w \leq t - b$

The proof of Proposition 4 is obtained by simplifying our proof of Proposition 2, Section 4. The simplifications are the following: (1) the size of the set F_r is fixed to 0, (2) in all partial runs we remove any step of the $reader_1$ and (3) we reduce the number of partial runs used in the proof. Such a simplified proof is similar to the proof of [1] with the following differences: (a) our proof considers optimally resilient implementations and (b) our proof distinguishes resilience thresholds b and t. In the following, we first define a SWSR robust safe storage and then we give the complete proof of Proposition 4.

An algorithm *implements* a robust safe storage if every run of the algorithm satisfies *wait-freedom* and *safeness* properties. Here we give a definition of safeness for a SWSR safe storage.

In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. Denote by wr_k the kth WRITE in a run ($k \geq 1$), and by val_k the value written by the kth WRITE. Let $val_0 = \perp$. We say that a partial run satisfies *safeness* if the following property holds: if a READ rd is contention-free and it succeeds some WRITE wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$.

We proceed by proving Proposition 4.

Proof. Let I be the implementation that satisfies properties (1) and (2) of Proposition 4, such that $f_w > t - b$. Since in our model at most t servers fail in any run, we assume $f_w \leq t$. Hence, we discuss only the case where $b > 0$. 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 server and at most b servers, B_2 (resp., T_1) that contains at most b (resp., t) servers, and F_w that contains exactly f_w 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, are empty.

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 terminates after accessing every correct server (B_1, B_2 and T_1) at most once. Hence, wr_1 completes after receiving responses to the first message sent to correct servers (B_1, B_2 and T_1). 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 while 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.

Now consider a partial run r_2 slightly different from r'_1 in which the writer fails during wr_1 such that the messages sent by the writer in wr_1 are delivered only to B_1 - other servers do not receive any message from the writer. In r_2, T_1 crashes at the beginning of the partial run. Assume that the writer fails at time $t' > t_{fail_w}$. Let rd be a READ operation invoked by *reader* after time t_{fail_w}. Since the only faulty server in r_3 is T_1, rd eventually completes, possibly after the messages in X_{wr_1} are delivered. Assume rd completes at time t_2 and returns v_R. Note, however, that all messages in X_{wr_1} in r_2 are causally preceded exclusively by the messages sent by B_1 (which is the only server to receive a message from the writer during wr_1).

Let r_3 be the partial run identical to r'_1 except that in r_3: (1) a READ operation rd is invoked by the correct reader *reader* at t' (as in r_2), (2) due to asynchrony all messages sent by T_1 to *reader* and the other servers are delayed until after t_2 and (3) at the beginning of r_3, B_2 fails maliciously: B_2 plays according to the protocol with respect to the writer, but to all other servers and *reader*, B_2 plays like it never received any message from the writer; otherwise, B_2 respects the protocol. Note that *reader* and servers F_w, and B_1 cannot distinguish r_3 from r_2 and, hence, rd terminates in r_3 at time t_2 (as in r_2) and returns v_R. By atomicity, as rd_1 precedes wr_1, v_R must equal v_1.

Now consider partial run r_4, identical to r_2, except that in r_4: (1) wr_1 is never invoked, (2) B_1 fails maliciously at the beginning of the r_4, forges that it received a wr_1 message from the writer and, otherwise, B_1 sends the same messages as in r_3, and (3) T_1 is not faulty in r_4, but, due to asynchrony, all messages sent by T_1 to *reader* and the other servers are delayed until after t_2. The reader and the servers F_w and B_2 cannot distinguish r_4 from r_2, so rd completes at time t_2 and returns v_R, i.e., v_1. However, by safeness, in r_4 rd must return $\perp$. Since $v_1 \neq \perp$, r_4 violates safeness. $\square$

C An inherent price of *lucky* reads

C.1 Preliminaries

In this Appendix, we answer the general question of the minimal number of servers under which an atomic storage implementation can have (1) *fast lucky* READS whenever at most $f_r, f_r \leq t$ servers fail, as well as (2) WRITES that always complete in at most two communication round-trips (we simply say that an implementation features *2-round WRITES*). Namely, such implementations are possible if and only if the total number of servers is at least $2t + b + min(b, f_r) + 1$.

The above result applies to a restricted communication model, we call *data-centric*, in which servers send messages only in response to clients' messages, i.e., servers do not communicate with other servers, nor they send unsolicited messages. In Appendix C.2, we modify our model of Section 2 to formally define the restricted, *data-centric* model.

We proceed by proving, in Appendix C.3 that $2t + b + min(b, f_r)$ servers are not enough to enable an atomic storage implementation that would feature *fast lucky* READS whenever up to f_r servers fail as well as *2-round WRITES*. The main idea underlying the proof of this lower bound is the observation that we can build, starting from a run of the algorithm with a complete write followed by a lucky read, a series of indistinguishable runs where we manage to completely erase the steps taken in the second round of the write, hence leaving a subsequent, unlucky reader, with the impossible task of determining a value simply by looking at the steps taken in the first round of the write. These can be forged by malicious servers, in case there is an insufficient total number of servers.

Finally, we prove in Appendix C.4 that $2t+b+min(b, f)+1$ servers are sufficient. The intuition of why an additional server circumvents the impossibility, which is thus key underlying our matching algorithm, is the following. Basically, we ensure that, (a) in case $b \geq f$, at least one correct server will report to the unlucky reader those steps taken in the second round of the write that the preceding lucky reader observed, and, (b) in case $b < f$, that at least $b + 1$ correct servers will report the steps taken in the first round of the write, overwhelming malicious servers. The matching algorithm that features both *fast lucky* READS despite the failure of f_r servers and *2-round WRITES* is very similar to our algorithm of Section 3.

C.2 Restricted communication model

To formalize the model in which servers send messages only in response to the clients' messages we modify our model of Section 2 as follows.

A non-malicious server 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.

In such a modified model, we define the notion of a communication round-trip as follows.

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 servers. This is indeed without loss of generality because we can simply model the fact that messages are not sent to certain servers by having these servers not change their state or reply.
2. Servers, on receiving such a message, reply to the reader (resp., writer) before receiving any other messages (as dictated by our modified model).
3. When the invoking client receives a sufficient number of such replies, a round rnd completes.

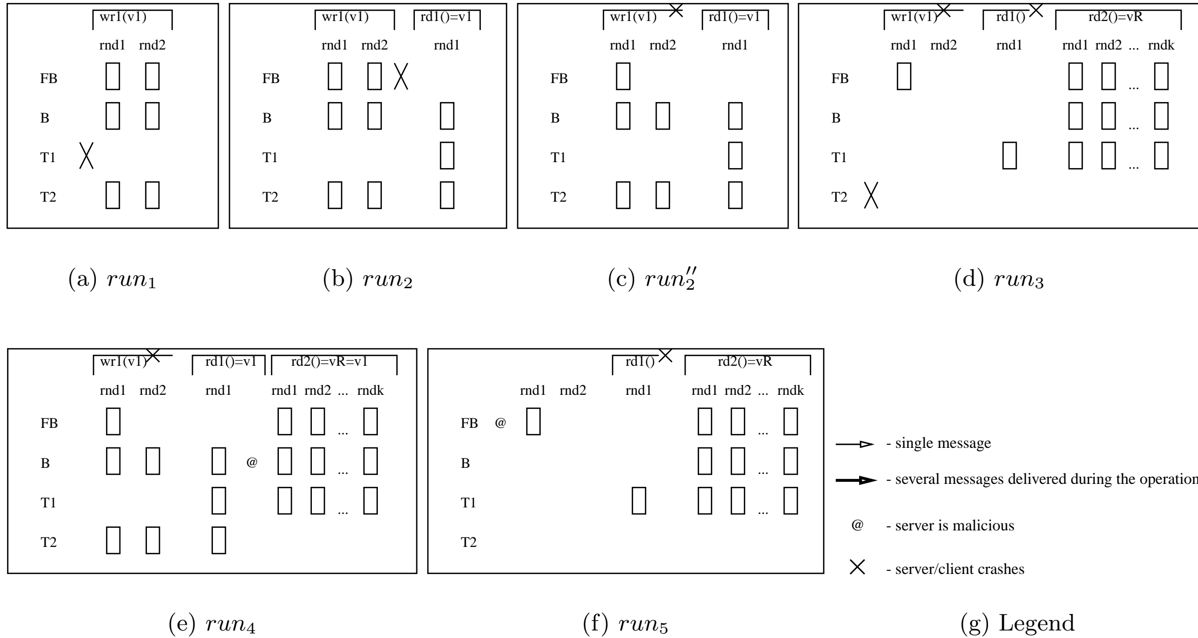
Apparently, in this model, every (complete) operation completes in a finite number of rounds.

C.3 Lower bound

Proposition 5. Let I be any implementation of a SWMR atomic wait-free storage, with the following properties: (1) in any partial run in which at most f_r servers fail, every *lucky* READ operation is *fast*, and (2) in any partial run, every (complete) WRITE operation completes in at most two communication round-trips. Then, $S \geq 2t + b + min(b, f_r) + 1$.

Proof. Suppose, by contradiction, that there is an atomic implementation I that satisfies Proposition 5 such that $S \leq 2t + b + min(b, f)$.

We partition the set of servers into four distinct blocks, denoted by T_1 and T_2, each of size at most t, B of size at most b, and FB of size at most $min(b, f)$. We consider only the case in which $f > 0, b > 0$, since, in case $f = 0$ or $b = 0$, it is not difficult to see that Proposition 5 holds, as the number of servers for any implementation I must conform with the optimal resilience lower bound of $S \geq 2t + b + 1$ [21]. Therefore, without loss of generality, we can assume that each of the blocks T_1, T_2, B and FB contains at least one server.



(a) run_1 (b) run_2 (c) run_2'' (d) run_3

(e) run_4 (f) run_5 (g) Legend

Fig. 5. Illustration of the runs used in the proof of Proposition 5

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

- Let run_1 be the run in which all servers are correct except T_1, which crashes at the beginning of the run. Furthermore, let wr_1 be the WRITE operation invoked by the correct writer in run_1 to write a value $v_1 \neq \perp$ in the storage and no other operation is invoked in run_1. By our assumption on I, wr_1 completes in run_1, say at time t_1 in at most two communication rounds. Therefore, wr_1 skips T_1, and completes (at latest) after the writer receives the replies in round 2 from correct servers (FB, B, and T_2).
- Let run_1' be the partial run that ends at t_1, such that run_1' is identical to run_1 up to time t_1, except that in run_1' server 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_1 from run'_1, wr_1 skips T_1 and completes in run'_1 at time t_1.

- Let the partial run run_2 extend run'_1 such that: (1) FB crashes at t_1, (2) rd_1 is a *lucky* READ operation invoked by the correct reader r_1 after t_1, (3) according to the proposition, rd_1 is *fast* (since $|FB| \leq f$) and completes at time t_2, skipping FB, (4) no additional operation is invoked in run_2, (5) run_2 ends at t_2, (6) all messages that were in transit in run'_1 remain in transit in run_2.
- Let run'_2 be the partial run identical to run_2 except that in run'_2 server FB does not crash, but, due to asynchrony, the message sent by r_1 to FB during rd_1 remains in transit. Since r_1 and all servers, except FB, cannot distinguish run_2 from run'_2, rd_1 completes in run'_2 at time t_2 (note that, both in run_2 and run'_2, rd_1 skips FB).
- Let run''_2 be the partial run that is identical to run'_2 except that, in run''_2: (1) the writer crashes during wr_1 and its round 2 message skips FB (and T_1). Since r_1 and all servers, except FB, cannot distinguish run'_2 from run''_2, rd_1 completes in run'_2 at time t_2 (note that, both in run'_2 and run''_2, rd_1 skips FB).
- Consider now a partial run run_3 slightly different from run''_2 in which the writer (resp., r_1) crashes during the round 1 of wr_1 (resp., rd_1) such that the round 1 messages sent by the writer (resp., r_1) in wr_1 (resp., rd_1) skip $\{B, T_1, T_2\}$ (resp., $\{FB, B, T_2\}$). We refer to the state of FB *after* sending the reply to the round 1 message of wr_1 as to σ_1. In run_3, T_2 crashes at the beginning of the partial run. Assume that the writer crashes at time t_{fail_w} and that r_1 crashes at time $t_{fail_r} > t_{fail_w}$. Let rd_2 be a READ operation invoked by the correct reader $r_2 \neq r_1$ at time $t'_3 > max(t_{fail_r}, t_2)$. Since the only (crash) faulty server in run_3 is T_2, rd_2 eventually completes, skipping T_2. However, rd_2 is not necessarily *fast*. Assume rd_2 completes at time t_3 after k communication rounds and returns v_R.
- Let run_4 be a partial run identical to run''_2 except that in run_4: (1) a READ operation rd_2 is invoked by the correct reader r_2 at t'_3 (as in run_3), (2) due to asynchrony all messages sent by T_2 to r_2 are delayed until after t_3 (i.e., until after k^{th} round of rd_2) and (3) at the beginning of run_4, B fails maliciously: it forges its state at t_2 to σ_0 (the initial state of servers); otherwise, B respects the protocol (including with respect to the writer and the reader r_1). Note that r_2 and the servers FB and T_1 cannot distinguish run_4 from run_3 and, hence, rd_2 completes in run_4 at time t_3 (as in run_3), skips T_2 and returns v_R. On the other hand, r_1 cannot distinguish run_4 from run''_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 run_5, identical to run_3, except that in run_5: (1) wr_1 is never invoked, (2) FB fails maliciously at the beginning of run_5 (this is possible since $|FB| \leq b$) and forges its state to σ_1 (see run_3); otherwise, FB sends the same messages as in run_3, and (3) T_2 does not crash in r_5, but, due to asynchrony, all messages sent by T_2 to r_2 are delayed until after t_3 (i.e., k^{th} round of rd_2). The reader r_2 and the servers B and T_1 cannot distinguish run_5 from run_3, so rd_2 completes at time t_3, skips T_2 and returns v_R, i.e., v_1. However, by atomicity, in run_5, rd_2 must return $\perp$. Since $v_1 \neq \perp$, run_5 violates atomicity.

C.4 Algorithm

We prove our lower bound of Appendix C.3 tight by proving the following proposition.

Proposition 6. There is an implementation I of a SWMR atomic wait-free storage, with the following properties: (1) in any partial run in which at most f_r servers fail, every *lucky* READ operation is *fast*, (2) in any partial run, every (complete) WRITE operation completes in at most two communication round-trips and (3) $S = 2t + b + min(b, f_r) + 1$.

The algorithm that proves Proposition 6 is very similar to our algorithm of Section 3. The differences are the following:

- *WRITE operation.* (1) W phase takes always two communication rounds (without using the timer in any round), (2) servers do not maintain the variable vw and (3) the writer sends the field $frozen$ within the W message instead of PW message.
- *READ operation.* (1) Definition of the predicate $fast()$ is modified (to facilitate the absence of *fast* WRITEs and WRITES that complete in three rounds), and (2) *writeback* procedure takes only two communication rounds (following the pattern of the WRITE implementation).

The entire modified pseudocode of our algorithm is given in Figures 6, 7 and 8. In the following, in Appendix C.5, we prove that our modified algorithm satisfies Proposition 6.

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

WRITE(v) is {
3:  inc(ts);
4:  pw := ⟨ts, v⟩
5:  send PW⟨ts, pw, w⟩ to all servers
6:  wait for (reception of PW_ACK_i⟨ts, newread⟩ from S − t different servers s_i)
7:  freezevalues()
8:  w := ⟨ts, v⟩
9:   send W⟨2, ts, pw, frozen⟩ message to all servers
10:   frozen := ∅
11:   wait for reception of WRITE_ACK_i⟨2, ts⟩ message from S − t different servers s_i
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]⟩ }
```

Fig. 6. Proposition 6 algorithm: WRITE implementation (at the writer)

C.5 Correctness

We first prove atomicity and then we proceed to wait-freedom and complexity. Note that in order for rd to return a value $c.val$, it is necessary that $c \in C$ (lines 16-18, Fig. 7). Therefore, if some complete READ returns $c.val$ then c must satisfy one of the following predicates (line 16, Fig 7): (1) $safe(c) \wedge highCand(c)$, or (2) $safeFrozen(c)$. For simplicity of presentation, in the following, if c satisfies predicate (1), we simply say that *c is live*, and if c satisfies predicate (2), we say that *c is frozen.*

Recall that, in the following, we assume $S = 2t + b + min(b, f_r) + 1$.

Lemma 9. ***No-creation.*** *If a READ rd completes and returns some value v, then either v was written by some WRITE or v is the initial value $\perp$.*

Proof. Suppose by contradiction that some READ (by some reader r_j) returns a value that is neither $\perp$, nor written by some WRITE. In that case, let rd be the first READ (according to the global clock) to select such a value v at line 18, Fig. 7 at time t_v (it is not difficult to see that such a rd exists). Therefore, if any READ operation has written back a value up to t_v, this value must have

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: $fast(c) ::= (|\{i : w_i = c\}| \geq S - t - f_r)$
6: $invalidw(c) ::= |\{i : \exists c' : readLive(c', i) \wedge$
$\wedge (c'ts < c.ts \vee (c'.ts = c.ts \wedge c'.v \neq c.v))\}| \geq S - t$
7: $invalidpw(c) ::= |\{i : \exists c' : pw[i] = c' \wedge$
$\wedge (c'ts < c.ts \vee (c'.ts = c.ts \wedge c'.v \neq c.v))\}| \geq S - b - t$
8: $highCand(c) ::= \forall c' \forall i : (readLive(c', i) \wedge c'.ts \geq c.ts \wedge c' \neq c) \Rightarrow$
$\Rightarrow invalidw(c') \wedge invalidpw(c')$
9: $tsr := tsr_0$; $T := timer()$;

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

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

$writeback(c)$ is {
24: **for** round= 1 to 2 **do**
25: send $W\langle round, tsr, c \rangle$ message to all servers
26: **wait for** receive $WRITE_ACK_i\langle round, tsr \rangle$ from $S - t$ servers }

Fig. 7. Proposition 6 algorithm: READ implementation (reader r_j)

been $\bot$, or it has been written by some WRITE. Without loss of generality, assume that v is returned as $v = c.val$, where $c.ts = ts$. Then, according to the line 16, Fig. 7, for the timestamp value pair c either $safe(c)$ or $safeFrozen(c)$ hold, i.e., either $b+1$ servers have sent a $READ-ACK_i$ message containing c in either pw or w ($safe(c)$) fields, or $frozen_{r_j}$ ($safeFrozen(c)$) field, including at least one non-malicious server s_i. Note that non-malicious servers update their pw, w and $frozen_{r_j}$ fields only when they receive a PW or W message from the writer, or a writeback message (WB) from the reader. Since until time t_v no reader has sent any writeback message containing a value that is neither $\bot$ nor written by some WRITE, we conclude that c was written by some writer or it was never updated by s_i, so it is $\bot$. A contradiction. □

Lemma 10. ***No ambiguity.*** *No two non-malicious servers ever store different values with the same timestamp.*

Proof. Note that the writer never assigns different values to the same timestamp (lines 3-4, Fig. 6). By Lemma 9, when (if) writing back a value, the readers always write back a value along with the timestamp the writer assigned to it. Therefore, it is impossible that two non-malicious servers store different values with the same timestamp in their pw, w or $frozen_*$ variables. □

Lemma 11. ***Non-decreasing timestamps.*** *Non-malicious servers never replace a newer value with an older one in their pw and w fields.*

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

3: **upon** reception of a $PW\langle ts, pw', w'\rangle$ message from the writer **do**
4: update(pw, pw'); update(w, w')
5: $newread := \bigcup\langle r_j, ts_{r_j}\rangle$, for all r_j such that $ts_{r_j} > frozen_{r_j}.tsr$
6: send $PW_ACK_i\langle ts, newread\rangle$ to the writer

7: **upon** reception of a $READ\langle tsr', rnd'\rangle$ message from the reader r_j
8: **if** $(tsr' > ts_{r_j})$ **and** $(rnd' > 1)$ **then** $ts_{r_j} := tsr'$ **endif**
9: send $READ_ACK_i\langle tsr', rnd', pw, w, vw, frozen_{r_j}\rangle$ to the reader r_j

10: **upon** reception of a $WB\langle round, ts, c\rangle$ message from r_j **or** $W\langle round, ts, c, frozen\rangle$ message from the writer **do**
11: update(pw, c)
12: **if** $round > 1$ then update(w, c) **endif**
13: **if** $sender = writer$ **then** $\forall j : \langle r_j, pw'_j, tsr'_j\rangle \in frozen$ **do**
14: **if** $tsr'_j \geq ts_{r_j}$ **then** $frozen_{r_j}.tsr := tsr'_j$; $frozen_{r_j}.pw := pw'_j$ **endif**
15: send $WRITE_ACK_i\langle round, ts\rangle$ message to the sender

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

Fig. 8. Proposition 6 algorithm: code of server s_i

Proof. Obvious from server code inspection, Figure 8. □

Lemma 12. ***Frozen is concurrent.*** *If a complete READ rd returns c.val, such that c is frozen, then c is written by some WRITE wr concurrent with rd.*

Proof. We prove this lemma by contradiction. Suppose wr is not concurrent with rd. There are two possibilities: (1) wr precedes rd, and (2) rd precedes wr. Without loss of generality we consider the case where the reader r_j executes rd.

Consider case (1). Note that every READ by r_j has its distinct, monotonically increasing timestamp. Let the timestamp for rd be tsr_{rd}. A non-malicious server changes its ts_{r_j} variable only if it receives the $READ$ message from r_j containing a timestamp greater than ts_{r_j} (line 8, Fig. 8). Therefore, a non-malicious server sets ts_{r_j} to tsr_{rd} only once rd started. In addition, the writer "freezes" the value (the one written by wr) for tsr_{rd}, at the end of the PW round of wr (and then sends it within a W round of wr), only when the writer receives at least $b+1$ server $PW - ACK_i$ messages in wr that contain the pair $\langle r_j, tsr''\rangle$ in the $newread$ fields, where $tsr'' \geq tsr_{rd}$ and the $b+1$st highest value is tsr_{rd}. Among these $b+1$ tsr'' values, at least one is from a non-malicious server. Therefore, rd must have already started when the writer freezed the value for tsr_{rd}. This is a contradiction with the assumption that wr precedes rd.

Consider now case (2). Note that the writer sends the $\langle r_j, c, read - ts[r_j]\rangle$ (lines 9 and 15, Fig. 6) in the field $frozen$ of the W message of wr. Note also that rd by r_j returns $c.val$ such that c is frozen only if the latest copy of the variable $frozen_{r_j}$ of at least $b+1$ servers contains $frozen_{r_j}.pw = c$ and $frozen_{r_j}.tsr = tsr_{rd}$, where tsr_{rd} is a timestamp of rd. This includes at least one non-malicious server s_i that updates its $frozen_{r_j}$ variable only upon receiving the W message sent during wr. Therefore, rd has not yet completed when wr was invoked. A contradiction. □

Lemma 13. ***Locking a pw value.*** *If a set X of at least $t+b+1$ non-malicious servers set their local variable pw such that $pw.ts \geq pw'.ts$ by time t', then a complete READ rd invoked after t' cannot return a value $pw''.val$ such that $pw''.ts < pw'.ts$.*

Proof. First, consider the case where pw'' is *live*. Every server in X stores pw such that $pw.ts \geq pw'.ts$ before rd is invoked. In addition, the timestamps of pw variables at non-malicious servers

are non-decreasing (Lemma 11). The response of each server in X to rd, if any, contains pw, such that $pw.ts \geq pw'.ts$. Since a reader in every round of READ awaits responses from $S - t$ servers (line 17, Fig. 7), at least $b + 1 \geq 1$ servers from the set X will respond to every round of rd. Let c be the smallest timestamp-value pair contained in the pw field of some response to rd from a non-malicious server s_i, for which $c.ts \geq pw'.ts$. We prove, by contradiction, that c is not *invalidpw*. By definition of *invalidpw*, a set Y of at least $S - b - t$ servers must have responded with values c' in their pw fields, such that $c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Since X contains only non-malicious servers and $|X| \geq t + b + 1$, and as $|X| + |Y| > S$, X and Y intersect in at least 1 (non-malicious) servers, so at least one of the responses from Y is from the non-malicious server s_j that belongs to the set X. Therefore, $pw_j.ts \geq c.ts$ and, thus, $pw_j.ts = c.ts \wedge pw_j.val \neq c.val$. However, as both s_i and s_j are non-malicious, this is impossible (Lemma 10). A contradiction.

Therefore, c where $c.ts \geq pw'.ts$ is not *invalidpw* and READ rd can not return any pw'' such that: (1) pw'' is live, and (2) $pw''.ts < pw'.ts$.

Now suppose that pw'' is *frozen*. Let $pw_1.ts$ be the lowest timestamp of every $pw.ts\ gepw'.ts$ in X. Since non-malicious servers store only values written by some WRITE, we conclude that a WRITE with a timestamp $pw_1.ts$ was invoked before rd. If rd returns pw'' such that $pw''.ts < pw'.ts \leq pw_1.ts$, then a WRITE with a timestamp $pw''.ts$ precedes rd. This violates Lemma 12. A contradiction. □

Lemma 14. ***Locking a w value.*** *If a set of X of at least $t + 1$ non-malicious servers have set their local variables pw and w such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$ by time t', a complete READ rd invoked after t cannot return a value $w''.val$ such that $w''.ts < w'.ts$.*

Proof. First consider the case where pw'' is *live*. Every server in X stores pw and w such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$ before rd is invoked. In addition, the timestamps of variables pw and w at non-malicious servers are non-decreasing (Lemma 11), each server in X, if it responds to rd, its response in every round will contain pw and w, such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$. Since a reader in every round of READ awaits responses from $S-t$ servers (line 17, Fig. 7), at least 1 server from the set X will respond to every round of rd. Let c be the smallest timestamp-value pair returned in the pw or w field by a non-malicious server s_i, for which $c.ts \geq w'.ts$. We prove, by contradiction, that c is not *invalidw*. By definition of *invalidw*, a set Y of at least $S-t$ servers must have responded with values c' in their pw or w fields, such that $c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Since X contains only non-malicious servers and $|X| \geq t+1$, $|X| + |Y| > S$, so at least one of the responses from Y is from the non-malicious server s_j that belongs to the set X. Therefore, $pw_j.ts \geq c.ts$ and $w_j.ts \geq c.ts$ and, thus, $pw_j.ts = c.ts \wedge pw_j.val \neq c.val$ or $w_j.ts = c.ts \wedge w_j.val \neq c.val$. However, as both s_i and s_j are non-malicious, this is impossible (Lemma 10). A contradiction.

Therefore, c where $c.ts \geq w'.ts$ is not *invalidpw* and READ rd cannot return any w'' such that: (1) w'' is live, and (2) $w''.ts < w'.ts$.

Now suppose that pw'' is *frozen*. Let $w_1.ts$ be the lowest timestamp of every $w.ts\ gew'.ts$ in X. Since non-malicious servers only store values written by some WRITE, we conclude that a WRITE with a timestamp $w_1.ts$ was invoked before rd. If rd returns w'' such that $w''.ts < w'.ts \leq w_1.ts$, then a WRITE with a timestamp $w''.ts$ precedes rd. This violates Lemma 12. A contradiction. □

Lemma 15. ***Atomicity of READ with respect to WRITEs.*** *If a READ rd is complete and it succeeds some complete $wr = WRITE(v)$, then rd does not return a value older than v.*

Proof. First note that the timestamps monotonically increase at the writer and all the values that any reader returns (and, hence, all the values that it writes back) are written by the writer. Therefore, timestamps associated to the values by the writer in line 4, Fig 6, order the values that readers return. Now, we show that rd does not return $c.val$, such that $c.ts < ts$, where ts is

associated to v in line 4, Fig 6 of $wr = WRITE(v)$. We consider two cases: (1) c is *live* and (2) c is *frozen*.

1. $safe(c) \wedge highCand(c)$ (c is live). In this case, $\langle ts, v\rangle$ is written to pw and w fields of at least $S - t = t + b + min(b, f_r) + 1$ servers, out of which a set X, of size at least $S - t - b = t + min(b, f_r) + 1$, contains only non-malicious servers. We can apply Lemma 14 ($w'.ts$ being ts), and conclude that rd does not return a value with a timestamp less than ts, i.e., a value older than v. Therefore, no c' such that $c'.ts < ts$ can satisfy the $highCand$ predicate, and no such c can be returned as live. A contradiction.
2. $safeFrozen(c)$ (c is frozen). Due to lemma 12, c has been written by some WRITE wr' concurrent with rd. Since (1) the writer assigns non-decreasing timestamps to values, (2) wr precedes rd and (3) rd is concurrent with wr', we conclude that wr precedes wr' and $c.ts > ts$. □

Lemma 16. ***READ hierarchy.*** *If a READ rd is complete and it succeeds some complete READ rd' that returns v', then rd does not return a value older than v'.*

Proof. Note that a complete READ rd does not write back a value $c.val$ it returns in case rd is *fast*, i.e., in case rd selects c at line 18, Fig. 7 at the end of first round ($rnd = 1$), and if $fast(c)$ holds. We now show that rd does not return $c.val$, such that $c.ts < ts'$, where ts' is associated to v' in line 4, Fig. 6 of the WRITE that wrote v'. We consider two cases: the first where rd' is *fast* and the second in which rd' is *slow* and writesback $c' = \langle ts', v'\rangle$ (lines 24-26, Fig. 7).

1. rd' is *fast*. In this case, in rd' a reader has received at least $S - t - f_r = t + b + min(b - f_r, 0) + 1 \geq b + 1$ different server responses that contain c' in their w fields. We consider two cases: (a) $b < f_r$ and (b) $b \geq f_r$.

   a. $b < f_r$. In this case, in rd' a reader has received at least $b + 1$ different server responses that contain c' in their w fields. This includes at least one non-malicious server. As malicious servers change their w fields in the W round (or the second round of WB - write back by some reader) only if the PW round (or the first round of WB) has already completed, i.e., if at least $S - t$ servers changed their pw field to c'. This includes a set X of at least $S - t - b = t + b + 1$ servers non-malicious servers. Applying Lemma 13, we conclude that rd returns a value $c.val$ such that $c.ts \geq c'.ts$.

   b. $b \geq f_r$. In this case, in rd' a reader has received at least $t + b + 1$ different server responses that contain c' in their w fields. This includes at least $t + 1$ non-malicious servers, that have set their pw and w variables to c'. Hence, we can apply Lemma 14, and conclude that rd does not return a value with a timestamp less than $c'.ts$, i.e., older than $c'.val$.
2. rd' writesback the value. In this case, in rd' the reader has completed writing back the value $c'.val$ and a set X of at least $S - t - b = t + min(b, f_r) + 1$ non-malicious servers have all set their pw and w variables to c' at latest during rd', i.e., before rd is invoked. Hence, we can apply Lemma 14, and conclude that rd does not return a value with a timestamp less than $c'.ts$, i.e., older than $c'.val$. □

Theorem 6. ***Atomicity.*** *The algorithm in Figures 6, 7 and 8 is atomic.*

Proof. By Lemmas 9, 15 and 16. □

We proceed by proving the wait-freedom property.

Theorem 7. ***Wait-freedom.*** *The algorithm in Figures 6, 7 and 8 is wait-free.*

Proof. The argument for the wait-freedom of a WRITE operation is based on the assumption that there are at most t faulty servers. In every round of a WRITE, the writer waits for at most $S-t$ valid server responses that the writer is guaranteed to receive eventually.

The argument for the wait-freedom of a READ operation is slightly more involved. We prove that, in every run, every READ operation rd invoked by the correct reader completes. Recall that, by our model, in any run at most t servers can fail, out of which at most b can be malicious.

We distinguish two cases: (a) the case where the writer issues a finite number of WRITE operations in the run, and (b) the case where there is an infinite (unbounded) number of WRITE operations in the run.

Case (a) - Finite number of WRITES. In this case there is a WRITE operation with the highest timestamp. Let wr denote the last complete WRITE operation that writes v with timestamp ts (or $v = \perp$, $ts = ts_0$ if there is none. We denote with wr' a possible later (incomplete) WRITE that writes v' with ts'.

Assume, by contradiction, that READ rd never returns a value. Then, rd invokes rounds on all correct servers, sending $\langle READ \rangle$ messages infinitely many times. We distinguish two cases: (a.1) value v' is never pre-written (written in the pw fields) into more than b correct servers and (a.2) there is a time t at which wr' is pre-written into $b+1$ or more correct servers.

In case (a.1) let t be the time at which last correct server changed its pw to $\langle ts', v' \rangle$ (by wr' or an incomplete writeback that sends a WB message with a same timestamp-value pair). In both cases, let $t' > t$ be the time at which rd received at least one response from every correct server sent after t.

Consider case (a.1). First we prove that $\langle ts, v \rangle$ is $safe$. Since $\langle ts, v \rangle$ was written to pw and w fields of at least $S-t$ servers, out of which at least $S-2t = b + min(b, f_r) + 1$ are correct. Out of these, no server ever changes its value w (as this would require that c' is pre-written in at least $S-2t = b+min(b, f_r)+1 > b$ correct servers). Hence, from time t' onward, $\langle ts, v \rangle$ appears at least $b+1$ times in $w[*]$ and is safe.

Moreover, there are at least $S-t$ responses in $w[*]$, from correct servers, with either $\langle ts, v \rangle$ or with a timestamp smaller than ts. Therefore, every timestamp-value pair c, such that $c.ts > ts \vee (c.ts = ts \wedge c.val \neq v)$ is $invalidw$. Finally, there are at least $S-t-b$ responses from correct servers in $pw[*]$ with either $\langle ts, v \rangle$ or with a timestamp smaller than ts. Therefore, every timestamp-value pair c, such that $c.ts > ts \vee (c.ts = ts \wedge c.val \neq v)$ is $invalidpw$. Thus, at the next iteration, $highCand(\langle ts, v \rangle)$ and $safe(\langle ts, v \rangle)$ hold, $\langle ts, v \rangle \in C$ and rd returns, a contradiction.

Now consider case (a.2), where v' is pre-written (by a WRITE or some READ) into $b+1$ or more correct servers. Then, after t', $\langle ts', v' \rangle$ appears at least $b+1$ times in $pw[*]$ and is safe. It is not difficult to see, as no subsequent valid value is present in the system, such that for every pair c'' such that $c''.ts > c'.ts \vee (c''.ts = c'.ts \wedge c''.val \neq c'.val)$ $invalidw(c'')$ and $invalidpw(c'')$ holds. Thus in the next iteration, $\langle ts', v' \rangle \in C$ and READ returns: a contradiction.

Case (b) - Infinite number of WRITES. Note that, in this case, the writer is correct. Suppose, by contradiction, that rd never completes. Let tsr_j be the timestamp of rd (at the correct reader r_j). Note that the reader r_j never invokes a READ rd' with a timestamp $tsr'_j > tsr_j$, and therefore, the writer cannot freeze a value (for the reader r_j) for a READ with a higher timestamp than tsr_j). Since r_j is correct, it invokes in rd an infinite number of rounds (rounds do not block as in every round the reader awaits $S-t$ responses). Eventually, r_j writes tsr_j to every correct server. Since there are $S-t = t+b+min(b, f_r)+1$ correct servers, in the next WRITE round (as there is an infinite number of WRITES, the writer executes an infinite number of WRITE rounds), the writer reads tsr_j from at least $b+min(b, f_r)+1$ of these servers in the PW round, freezes the current pw value (pw') for the reader r_j (it is not difficult to see that, for a particular read timestamp tsr_j,

the writer freezes at most one value) and sends it within the $W.frozen$ field in the W round of that very same write. Since (1) the channels are reliable, (2) servers take into account $W.frozen$ fields of the old W messages of the writer, and (3) the writer is correct, eventually, every correct server stores $frozen_{r_j} = \langle pw', tsr_j \rangle$. Since there are $S - t = t + b + min(b, f_r) + 1$ correct servers, in the next READ iteration, r_j reads the same $frozen_{r_j} = \langle pw', tsr_j \rangle$ from at least $b + min(b, f_r) + 1$ correct servers, pw' becomes $safeFrozen$ and rd terminates. A contradiction. □

Now we prove that our modified algorithm of Appendix C.4 has the following properties: (1) in any partial run in which at most f_r servers fail, every *lucky* READ operation is *fast*, (2) in any partial run, every (complete) WRITE operation completes in at most two communication round-trips. Since the property (2) is immediate, we prove that, if at most f_r servers are faulty, every *lucky* READ (i.e., the READ that is synchronous and contention-free) is *fast*.

Theorem 8. ***Fast READS.*** *In the algorithm of Figures 6, 7 and 8, if a READ operation rd is* lucky *and at most f_r servers fail by the completion of rd, then rd is* fast.

Proof. First, suppose that the last (complete) WRITE (wr) that precedes rd writes a timestamp-value pair c (or $c = \langle ts_0, \perp \rangle$ if there is no such WRITE). Since rd is contention-free, no other WRITE is invoked before the completion of rd, and, hence, no other value stored by a correct server has a timestamp higher than $c.ts$ during the execution of rd.

Hence, c was written in the pw and w fields of at least $S - t = t + b + min(b, f_r) + 1$ servers out of which at least $S - t - f_r = t + b + min(b - f_r, 0) + 1 \geq b + 1$ are correct and respond with a valid $READ - ACK$ message in the first round of rd. Therefore, at the end of round 1, $safe(c)$ and $fast(c)$ (line 5, Fig. 7) hold. Let c' be any timestamp-value pair such that $\exists i, readLive(c', i), c'.ts \geq c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Then, as there are no WRITES after wr that are concurrent with rd, and as rd is synchronous, all $S - f_r \geq S - t$ correct servers respond before the expiration of the timer (and, hence, before the end of round 1) with timestamps less than $c'.ts$ or with a timestamp $c'.val = c.ts$ and a value $c.val \neq c'.val$. Therefore, at the end of round 1, for every such a timestamp-value pair c', the predicates $invalidw(c')$ and $invalidpw(c')$ hold. Finally, predicate $highCand(c)$ is true at the end of round 1 and rd returns $c.val$. □

D Regular Implementation

D.1 Preliminaries

In this appendix, we show how our SWMR atomic storage implementation can be transformed into the SWMR regular storage implementation. An algorithm *implements* a robust regular storage if every run of the algorithm satisfies *wait-freedom* and *regularity* properties. Here we give a definition of regularity for a SWMR regular storage.

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

The advantages of our regular storage implementation over our atomic one are the following: (1) it tolerates an arbitrary number of malicious readers and (2) the number of actual server failures *every lucky* operation can tolerate while still being *fast* is boosted to the maximum: in our regular implementation every *lucky* WRITE is *fast* despite the failure of up to $f_w = t - b$ servers[9] and every *lucky* READ is *fast* despite the failure of $f_r = t$ servers.

In the following we prove:

Proposition 7. There is an optimally resilient implementation I of a SWMR robust regular storage, such that: (1) in any partial run in which at most $t - b$ servers fail, every *lucky* WRITE operation is *fast*, (2) in any partial run in which at most t servers fail, every *lucky* READ operation is *fast*, and (3) I tolerates arbitrary number of malicious readers.

D.2 Modifications to the Atomic Implementation

The algorithm A that proves Proposition 7, is obtained from our algorithm of Proposition 1, Section 3 as follows: (1) W phase of WRITE operation takes only one instead of two rounds (line 9, Fig. 1), (2) in the READ emulation, the writeback procedure is removed (lines 21 and 26-28, Fig. 2) and (3) servers ignore every WB message sent by some reader r_j. Note that in line 8, Figure 1, the writer now checks if it has received $PW - ACK$ messages from $S - f_w = t + 2b + 1$ servers.

In the following we sketch the correctness proof for our regular implementation A.

D.3 Correctness

Lemma 17. *In our regular implementation A, non-malicious servers only store values written by the writer or $\langle ts_0, bot \rangle$ in their pw, w and $frozen_*$ variables.*

Proof. Obvious from the inspection of server code of Figure 3, as non-malicious servers ignore WB messages sent by readers (see Appendix D.2). □

Lemma 18. *No-creation.* *If a READ rd completes and returns some value v then either v was written by some previous WRITE or v is the initial value ⊥.*

Proof. Suppose that there is a READ rd that returns a value v that was not the initial value $\perp$ nor written by some WRITE. Without loss of generality, assume that v is returned as $v = c.val$,

[9] This is optimal, as proved in Proposition 3, Appendix A.

where $c.ts = ts$. Then, according to line 18, Fig. 2, for the timestamp value pair c, either $safe(c)$ or $safeFrozen(c)$ hold, i.e., $b+1$ servers have sent a message containing c in their either pw or w ($safe(c)$), or $frozen$ ($safeFrozen(c)$) fields, including at least one non-malicious server s_i. Since non-malicious servers update their pw, w and $frozen$ fields only when they receive a PW or W message from the writer (Lemma 17), we conclude that c was written by some writer or it was never updated by s_i, so it is $\perp$. A contradiction. □

The proofs of the following lemmas follow the proofs of the respective lemmas for the atomic implementation from Appendix **??**.

Lemma 19. ***Non-decreasing timestamps.*** *Non-malicious servers never replace a newer value with an older one in their pw or w.*

Lemma 20. ***Frozen is concurrent.*** *If a complete READ rd by a correct reader r_j returns $c.val$, such that c is frozen, then c is written by some WRITE wr concurrent with rd.*

Lemma 21. ***Locking a pw value.*** *If a set of X of at least $t+b+1$ non-malicious servers set their local variable pw such that $pw.ts \geq pw'.ts$ at time t', then a complete READ rd invoked by a correct reader r_j after t' cannot return a value $pw''.val$ such that $pw''.ts < pw'.ts$.*

Lemma 22. ***Locking a w value.*** *If a set of X of at least $t+1$ non-malicious servers have set their local variables pw and w such that $pw.ts \geq w'.ts$ and $w.ts \geq w'.ts$ at time t', a complete READ rd invoked by a correct reader r_j after t cannot return a value $w''.val$ such that $w''.ts < w'.ts$.*

Lemma 23. *If a READ rd by a correct reader r_j is complete and it succeeds some complete $wr = WRITE(v)$, then rd does not return a value older than v.*

Theorem 9. ***(Wait-freedom.)*** *The modified algorithm A of Appendix D.2 is wait-free.*

Regularity is proven using the above lemmas.

Theorem 10. ***(Regularity.)*** *The modified algorithm A of Appendix D.2 is regular.*

Proof. By Lemma 18 and Lemma 23. □

We give detailed proofs of the *fast* WRITE and *fast* READ properties of our regular implementation.

Theorem 11. ***(Fast Writes.)*** *Consider the modified algorithm A of Appendix D.2. If a complete WRITE is synchronous and at most $f_w = t - b$ servers fail by the completion of wr, then wr is* fast.

Proof. As at most $t-b$ servers are faulty and the WRITE is synchronous, the writer receives the response from at least $S - t + b = t + 2b + 1$ servers in the PW round before the timer *Timeout* expires (line 8, Fig. 1) and the WRITE completes in a single round-trip, before the second, W phase (round). □

Theorem 12. ***(Fast Reads.)*** *Consider the modified algorithm A of Appendix D.2. If a complete READ operation rd is synchronous and contention-free (i.e., if rd is* lucky*), then rd is* fast.

Proof. First, suppose that the last (complete) WRITE (wr) that precedes rd writes a timestamp-value pair c_{pw}. As rd is contention free, no other WRITE is invoked before the completion of rd, and, hence, no correct server stores a value with a timestamp higher than $c_{pw}.ts$ during the execution of rd.

First, suppose that wr is a *fast* WRITE. Thus, c_{pw} was written in the pw field of at least $S - f_w = 2b + t + 1$ servers out of which at least $S - f_w - t = 2b + 1$ are correct and respond with a valid $READ - ACK$ message in the first round of rd. Therefore, at the end of round 1, $safe(c_{pw})$ holds. Let c' be any timestamp-value pair such that $\exists i, readLive(c', i), c'.ts \geq c_{pw}.ts \vee (c'.ts = c_{pw}.ts \wedge c'.val \neq c_{pw}.val)$. Then, as there are no WRITES after wr that are concurrent with rd, and as the system is synchronous, all $S - t$ correct servers respond before the expiration of the timer (and, thus, round 1) with timestamps less than $c'.ts$ or with a timestamp $c'.val = c_{pw}.ts$ and a value $c_{pw}.val \neq c'.val$. Therefore, at the end of round 1, for every such timestamp-value pair c', the predicates $invalidw(c')$ and $invalidpw(c')$ hold. Finally, predicate $highCand(c_{pw})$ is true at the end of round 1 of rd and rd returns $c_{pw}.val$.

Now suppose that wr is a two-phase WRITE. Hence, c_{pw} was written in the pw and w fields of at least $S - t = b + t + 1$ servers out of which at least $S - 2t = b + 1$ are correct and respond with a valid $READ - ACK$ message in the first round of rd. Therefore, at the end of round 1, $safe(c_{pw})$ holds. Let c' be any timestamp-value pair such that $\exists i, readLive(c', i), c'.ts \geq c_{pw}.ts \vee (c'.ts = c_{pw}.ts \wedge c'.val \neq c_{pw}.val)$. Then, as there are no WRITES after wr that are concurrent with rd, and as the system is synchronous, all $S - t$ correct servers respond before the expiration of the timer (and, thus, round 1) with timestamps less than $c'.ts$ or with a timestamp $c'.val = c_{pw}.ts$ and a value $c_{pw}.val \neq c'.val$. Therefore, at the end of round 1, for every such timestamp-value pair c', the predicates $invalidw(c')$ and $invalidpw(c')$ hold. Finally, predicate $highCand(c_{pw})$ is true at the end of round 1 of rd and rd returns $c_{pw}.val$. □

E Contending With the Ghost

In this section, we prove the following theorem.

Theorem 13. *In the algorithm of Figures 1, 2 and 3, if the writer fails during an incomplete WRITE wr' at time T, then, for every reader r_j, at most three synchronous READ operations invoked by r_j after T are* slow.

However, we first prove a pair of lemmas.

Lemma 24. *If the writer fails at time T during $wr' = WRITE(v')$ and some* slow *READ rd invoked after T returns v', then every synchronous READ that succeeds rd is* fast.

Proof. Assume that a timestamp ts' is associated to v' in line 4 of wr'. By atomicity, every synchronous READ rd' that succeeds rd may return only v'. Moreover, as correct servers store only values written by some WRITE or $\perp$, and as wr' is the last WRITE invoked, no correct server ever stores (in its pw, w or vw variables) a value with a timestamp higher than ts'. Since rd is *slow*, it writesback the value and, when rd completes, a set X of at least $S - 2t = b + 1$ correct servers store $\langle ts', v' \rangle$ in their pw, w and vw fields. Since rd' is synchronous, all servers from X respond with $pw = vw = vw = \langle ts', v' \rangle$ in the first round of rd'. Therefore, at the end of round 1 of rd' predicates $safe(\langle ts', v' \rangle)$ and $fastvw(\langle ts', v' \rangle)$ hold. Moreover, it is not difficult to see that any timestamp value pair c such that $c.ts > ts'$ or $(c.ts = ts') \wedge (c.val \neq v')$ will be deemed $invalidpw$ and $invalidw$ at the end of round 1 of rd'. Therefore, $highCand(\langle ts', v' \rangle)$ holds and rd' returns v' without executing the writeback procedure, at the end of its first round. Therefore, rd' is *fast*. □

Lemma 25. *For every reader r_j, if (1) the writer fails at time T during $wr' = WRITE(v')$, (2) the last complete WRITE wr writes v*[10]*, and (3) some* slow *READ rd invoked after T returns v, then at most one synchronous READ rd' by r_j that succeeds rd is* slow *and returns v.*

Proof. Suppose, by contradiction, that there are two synchronous READS rd' and rd'' by r_j that succeed rd and that are both *slow* and return v, and let rd' precede rd''. Assume that a timestamp ts' (resp., ts) is associated to v' (resp., v[11]) in line 4 of wr' (resp., wr). By atomicity, every READ rd_1 by r_j that succeeds rd may return either v or v'. Moreover, as correct servers store only values written by some WRITE or $\perp$, and as wr' is the last WRITE invoked, no correct server ever stores (in its pw, w or vw variables) a value with a timestamp higher than ts'. Since rd is *slow*, it writesback the value and, when rd completes, a set X of at least $S - 2t = b + 1$ correct servers store either $\langle ts, v \rangle$ or $\langle ts', v' \rangle$ in their pw, w and vw fields.

We distinguish three cases: (1) the case where, at the end of the first round of rd', (a) at most b correct servers have responded with $pw_* = \langle ts', v' \rangle$ and (b) no correct server had responded with $w_* = \langle ts', v' \rangle$, (2) the case where, at the end of the first round of rd', $b + 1$ or more correct servers have responded with $pw_* = \langle ts', v' \rangle$, and (3) the case where, at the end of the first round of rd', some correct server has responded with $w_* = \langle ts', v' \rangle$.

Consider case (1). Since no correct server responds with $w_* = \langle ts', v' \rangle$ in the first round of rd', no correct server responds with $vw_* = \langle ts', v' \rangle$ in the first round of rd'. Since all correct servers respond in the first round of synchronous READ rd', all servers from the set X respond in the first round of rd' with $vw_* = \langle ts, v \rangle$. Therefore, at the end of round 1 of rd', predicates $safe(\langle ts, v \rangle)$ and $fastvw(\langle ts, v \rangle)$ hold. Consider any c such that $readLive(c)$ and $((c.ts > ts)$ or $((c.ts = ts) \wedge (c.val \neq v))$ holds. Since at least $S - t$ correct servers respond in the first round of rd' and at most b of those respond with $pw_* = \langle ts', v' \rangle$, $S - t - b$ correct servers respond with a

[10] In case such a wr does not exist, ve denotes $\perp$ as v.
[11] In case $v = \perp$, $ts = ts_0$.

timestamp-value pw such that $pw.ts < ts'$. Hence, it is not difficult to see that, at the end of round 1 of rd', for any c, $invalidpw(c)$ holds. Similarly, as no correct server responds with $w_* = \langle ts', v' \rangle$, for any c $invalidw(c)$ holds. Therefore, at the end of round 1 of rd', $highestValid(\langle ts, v \rangle)$ holds and v is returned at the end of round 1 of rd' and rd' is *fast*. A contradiction.

Consider case (2). Obviously, at the end of round 1 of rd' $safe(\langle ts', v' \rangle)$ holds. Moreover, by Lemma 2, the fact that no correct server ever stores a value with a timestamp higher than ts' and the fact that all (at least $S - t$) correct servers respond in the first round of rd', at the end of round 1 of rd' for any c such that $readLive(c)$ and $((c.ts > ts')$ or $((c.ts = ts') \wedge (c.val \neq v'))$ hold, $invalidpw(c)$ and $invalidw(c)$ also hold. Hence, rd' returns v'. A contradiction.

Finally, consider case (3). Since some correct server s_i responds in READ rd' with $w_* = \langle ts', v' \rangle$, and as rd' precedes rd'', by the time rd'' is invoked, at least $b + 1$ correct servers have updated their pw values to $pw_* = \langle ts', v' \rangle$. Since all correct servers respond in the first round of rd'', by the argument of case (2), rd'' returns v'. A contradiction. □

We proceed with the proof of Theorem 13.

Let wr be the last complete WRITE (if it exists) that writes $\langle ts, v \rangle$. If wr does not exist, we denote $\langle ts_0, \perp \rangle$ as $\langle ts, v \rangle$. Assume, by contradiction, that there are four *slow* synchronous READS invoked by some reader r_j after T, rd_1, rd_2, rd_3 and rd_4 (by order of precedence). By atomicity, all READS rd_1 to rd_4 return either v or v'. We distinguish two cases.
(1) rd_1 returns v. In this case, by Lemma 25, at least one of the READS rd_2 and rd_3 is either *fast* or returns v'. Since we assume that rd_2 and rd_3 are *slow*, at least one of the READS rd_2 or rd_3 returns v'. However, by Lemma 24, in any case READ rd_4 is *fast*. A contradiction.
(2) rd_1 returns v'. By Lemma 24, READS rd_2, rd_3 and rd_4 are *fast*. A contradiction.