Building Storage Registers from Crash-Recovery Processes*

Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch
HP Labs, Palo Alto, CA 94304

Abstract—This paper defines a new type of register, called a storage register, to represent blocks of a replicated logical volume built from a distributed collection of disks. We give a formal specification of storage registers and, in doing so, we extend linearizability to a crash-recovery model. Existing algorithms that implement registers on top of message-passing primitives typically assume a crash-stop failure model. Our work illustrates the difficulties in moving to a more general failure model and presents an efficient implementation of storage registers in a message-passing system with crash-recovery processes.

1 Introduction

This paper presents a new replication algorithm suitable for high-performance, highly available logical disk systems. We envisage a logical disk system architecture where a disk volume is striped and replicated across a number of *bricks*, or intelligent storage devices containing disks, a CPU, NVRAM and network cards.

We model each logical disk block as a read-write register. The bricks collectively emulate the functionality of a multi-writer, multi-reader register for each logical block. Our register implementation superficially resembles traditional atomic-register constructions for a message-passing model [1, 11, 12]. However, where these atomic register constructions assume a crash-stop model, our implementation handles process recovery as well. We extend existing work on shared-memory abstractions in the following directions.

- We develop an extension to linearizability [7] that enables reasoning about safety in a crash-recovery model.
- We define a new abstraction called a *storage register*. A storage register exploits the fact that concurrent access to the same logical block is extremely rare in real-world storage systems:[1] the read and write methods on a storage register are allowed to "abort" if they are invoked concurrently. We define precise liveness properties for a storage register that limit the possibility of perpetual abort to runs with perpetual concurrency.
- We give an efficient implementation of a storage register in an asynchronous message-passing model with process crash and recovery. Our algorithm ensures that if a process that starts an operation crashes before completing, that operation is linearized before any operations issued after process recovery. Moreover, our algorithm runs read operations more efficiently than existing atomic register constructions: in the normal, failure-free case, our algorithm completes reads in a single round-trip, as opposed to the two round-trips required by traditional algorithms.

Our algorithm is based on the notion of quorum [6], where any majority of replicas constitutes a quorum. Data consistency is always maintained and the algorithm makes progress whenever a majority of processes are able to communicate.

1.1 Recovery is the hard part

Bricks in a logical disk system do restart after they crash—the question is not *whether* to handle recovery, but *how*. We argue that disk replication demands support for a crash-recovery model. Adopting a crash-stop algorithm and implementing recovery as the addition of a new node is impractical, as it would require us to perform a full state transfer from existing

*Published as HP Labs Technical Report HPL-SSP-2003-14

[1] In fact, we have found no instance of concurrent accesses in any of the workloads we have studied. We discuss this issue further in Section 6.1.

nodes for the billions of blocks that a brick stores. In the following, we discuss the ramifications of reasoning about, and implementing, read-write registers in a crash-recovery model.

Addressing process recovery demands a new set of safety conditions for register abstractions. Traditional linearizability [7] allows only the last operation in each per-process history to be partial, because it assumes crash-stop processes. To support recovery, we define an extension to linearizability, in which a per-process history may contain an arbitrary number of partial operations. We demand that each (possibly partial) operation execute atomically: a partial operation may or may not take effect, but if it does, its place in the linearized history must be compatible with the inherent partial order of events in the distributed system.

Process recovery complicates the algorithm design, particularly regarding partial writes, where a replica r invokes a write operation and then crashes before the operation completes. A partial write should appear atomic, and it must appear to take effect, if at all, before r recovers.

Correctly handling partial writes in a crash-recovery model creates read-write conflicts that are not present in a crash-stop model. Consider the situation where process p starts a write operation, manages to store the new value only on a sub-majority of replicas before crashing, and then another process p' starts reading. Process p' may return the "old" value of the register, but it must prevent the partial write from taking effect after the read completes. Thus, the read must "abort" the partial write, and re-enforce the old value. But since a read cannot distinguish a slow write from a partial write, read operations may abort in-progress writes as well, which creates a read-write conflict.

The presence of read-write conflicts illustrates the difficulties in adapting the algorithms in [1, 11, 12] to a crash-recovery model—it is not sufficient to simply store key variables in stable storage. The presence of read-write conflicts also implies that a traditional atomic register [9] is not a good fit as a register abstraction for logical disk systems. To provide the semantics of an atomic read-write register in a crash-recovery model, we have to guarantee that conflicting read-write operations are always serialized, e.g., through some notion of leader election, by keeping additional control information, or by some other means. Paying the cost of employing these serialization mechanisms is particularly troublesome in the context of storage systems, because concurrent access to the same logical block is overwhelmingly rare. If we were to use conventional atomic registers, we would effectively pay for a conflict-resolution capability that we never need.

We thus define a new abstraction, called a *storage register*, that better reflects the properties of logical disk systems. Like an atomic register, a storage register supports read and write methods. Unlike an atomic register, however, these methods may return an error if invoked concurrently. If an operation returns an error, it is then up to the storage system client (e.g., a client-side SCSI driver) to retry the operation. An unsuccessful read has no effect on the register. An unsuccessful write may or may not update the register. However if a failed write does update the register, the effect is equivalent to that of a successful write.

1.2 Overview of the algorithm

We run instances of the algorithm independently for each logical disk block. The local property of linearizability [7] ensures that operations on a logical disk are linearizable, providing that operations on each logical block are linearizable. A read or write operation can be invoked by any replica of a register. We call this replica the *coordinator* of the operation (different operations can have different coordinators). The value stored in a register is a disk block (usually 1KB). Values are stored on disk, while timestamps can be kept in memory (NVRAM).

Our algorithm runs a "read" operation optimistically: without concurrent requests and failures, a read operation involves a single disk read at the coordinator and a single round-trip of messages between the coordinator and a majority of replicas. The messages contain timestamp information only, not the actual value. In the presence of concurrency and failures, a read operation may require two additional rounds of communication between the coordinator and a majority of replicas. This "slow read" phase involves a disk read and write at a majority of replicas. The write operation always involves two rounds of communication between the coordinator and a majority of replicas, with a disk write at a majority of replicas.

1.3 Related work

The algorithm in [1] pioneered the implementation of atomic registers in a message-passing model. However, the implementation assumes a crash-stop failure model and only supports a single writer. The algorithms in [11, 12] implement an atomic multi-writer, multi-reader register in a message-passing model. However, a read operation in these algorithms always involves a two-phase interaction with a quorum: one phase to read a ⟨value,timestamp⟩ pair from replicas, and the second phase to write the ⟨value,timestamp⟩ pair with the highest timestamp back to a quorum. In contrast, our optimistic read operation involves a single disk read only. Furthermore, these algorithms do not support stable storage and process recovery.

Like storage registers, Lamport's safe and regular registers [9] provide the semantics of an atomic register only in the absence of concurrency. However, operations on safe and regular register do not indicate, per se, if they are invoked concurrently, whereas operations on storage registers do (they return a distinct error value). The ΔRegister in [3] also returns a distinct value in the presence of concurrency. However, where ΔRegister allows non-concurrent processes to agree on a single value, a storage register allows non-concurrent processes to share data over time.

An alternative approach is to model each block of storage as a state machine [8], and use a total-order broadcast mechanism, such as Paxos [10], to apply read and write operations in the same order at all replicas. Fundamentally, since total-order broadcast is equivalent to consensus [2], it cannot be implemented in an asynchronous system [4], whereas atomicity can. Furthermore, total-order broadcast is more expensive both in terms of space and storage. For example, with Paxos, the delivery of any request (read or write) requires at least two rounds of communication and two disk I/Os.

Several recent algorithms [5, 13] demonstrate how to implement Paxos on top of storage-area networks with unreliable disks. The goal of these algorithms is to decouple processing and storage to (a) tolerate the crash of more processing units [5] and (b) to handle infinitely many processing units [13]. By contrast, our storage bricks contain both processing and storage, and our goal is to build better storage-area networks that provide the illusion of a single, highly-available disk.

1.4 Structure of the paper

In Section 2, we introduce the model that we assume for the algorithm. Section 3 defines storage registers and specifies the correctness criteria of our algorithm, including extended linearizability. We describe the algorithm in detail in Section 4 and sketch the correctness proof in Section 5. *To reviewers: the full proof appears in Appendix A.* Section 6 discusses practical issues that arise when applying our algorithm to real-world disk systems. Section 7 concludes and describes future work.

2 Model

We consider a system of n processes, $p_1, \ldots, p_n$, that collectively emulate a storage register. Processes fail by crashing, i.e., they do not behave maliciously. A process may recover later. A *correct* process is one that either never crashes or eventually stops crashing. A *faulty* process is a process that is not correct. We assume a majority of correct processes; i.e., $n \geq 2f+1$, where f is the number of faulty processes.

Processes are fully connected by a network and communicate by message passing. The system is asynchronous: there is no bound on message-transmission times, and there is no bound on the time it takes a process to execute a step in its algorithm. Channels, however, are assumed to have a fair-loss property:

FAIR LOSS**:** If a process p_i sends a message m an infinite number of times to a correct process p_j, then p_j receives m an infinite number of times.

NO CREATION**:** If a process p_i receives a message m from p_j, then p_j sent m.

Each process provides a non-blocking timestamp-generation primitive called newTS that returns a value in a totally ordered set (we use operators "$<$", "$>$", and "$=$" to compare timestamps). We assume that newTS satisfies the following properties:

UNIQUENESS: Any two invocations of newTS (possibly by different processes) return different timestamps.

ORDER: Successive invocations of newTS by a process p produce monotonically increasing timestamps.

PROGRESS: Assume that a process p_i receives a timestamp t from newTS. If a process p_j receives an infinite number of timestamps from newTS, then p_j will eventually receive a timestamp t' with $t' > t$.

We also assume that there is a smallest timestamp, called initialTS. For any value t returned by newTS, $t >$ initialTS. We discuss the practical aspects of implementing timestamp generation in Section 6.1.

Each process has persistent storage that survives crashes. The store(*var*) primitive atomically writes the current value of variable *var* to the persistent storage. When a process recovers, it automatically retrieves the most recently stored value for each variable and assigns this value to the variable.

A process has access to a timer to wait for an indeterminate period of time. Processes use timers to implement message re-transmission: after sending a message, a process uses a timer to wait before re-sending the message. The amount of time spent waiting affects the performance, but not the correctness, of the system. For correctness, we rely only on the fact that timers eventually expire.

3 Storage registers

A storage register is a read-write register that (a) can crash and recover, (b) behaves like a conventional atomic register when accessed in a non-concurrent manner, and (c) may abort concurrent operations. This section defines storage registers formally. We first introduce the concepts of operations and events in Section 3.1. Section 3.2 defines the safety of storage registers. We extend linearizability [7] in two ways to handle process recovery: (a) treat process crash as an unsuccessful completion of an operation, and (b) allow non-successful operations to leave the system in a non-deterministic yet well-defined state. Section 3.3 defines the liveness properties by combining the usual notion of non-blocking behavior with a constraint that limits the possibility of abort only to situations with concurrency.

3.1 Operations, events, and histories

We use the term *operation* to refer to a particular invocation of the read or write method on a register. A write operation takes a value and returns either OK or NOK. A read operation returns a value or NIL, which indicates an error. We use a special value $\perp$ to represent the initial state of a register. Thus, read may return $\perp$, but we assume that write is never invoked with $\perp$ as parameter. If a write operation returns OK, or if a read operation returns a non-NIL value (including $\perp$), the operation is said to be successful. If an operation returns NOK or NIL, the operation is unsuccessful.[2] For simplicity, we assume that a process issues read and write operations one at a time (but multiple processes can issue operations concurrently).

We use three types of events to model the execution of a system. An *invocation* event happens when a process starts a read or write operation. A *return* event happens when an operation completes. A *crash* event happens when a process crashes.

A run of our algorithm results in a *history*, or sequence of events. We assume that the ordering of events within a history complies with the inherent partial order of events in a distributed system [8]. For a history H and a process p, we define $H|p$ to be the history derived by extracting only the events that happen at p.

3.2 Linearizability

Roughly speaking, a history H is linearizable when (1) its events indicate that operations happen instantaneously in some total order, (2) this apparent total order of operations is compatible with the inherent partial order of events in a distributed system, and (3) the apparent total order is consistent with the sequential specification of our register [7].

Before we can use the formalism in [7] to define safety for a storage register, we need to fill two gaps. First, linearizability does not define the semantics of failed operations; we give a sequential specification

[2] As we define in Section 3.2.2, we also treat operations whose coordinator has crashed as being unsuccessful.

for such operations in the next section. Second, linearizability disallows the appearance of a partial operation in the middle of a per-process history. Section 3.2.2 defines a way to transform a history with an arbitrary number of partial operations into a well-formed one.

3.2.1 A sequential specification for storage registers

The sequential specification of a shared data object, such as a read-write register, captures the semantics of the object in the absence of concurrency and replication. For example, the sequential specification for a conventional read-write register simply demands that a read operation return the most recently written value. For storage registers, however, we cannot rely on the notion of "written value", because a write operation may not always complete successfully. Instead, we define the sequential specification of a storage register in the following manner:

- A successful write operation stores its value in the register.
- An unsuccessful write operation may or may not store its value in the register.
- A successful read operation returns either (a) the value most recently stored in the register or (b) $\perp$ if no value has been stored in the register.

3.2.2 Linearizing crash-recovery histories

For a history H to be well-formed, each per-process history $H|p$ must be sequential—that is, in $H|p$, each invocation event, except possibly the last, must be followed immediately by a return event [7]. In our per-process histories, any invocation event (not just the last) may be followed by a crash event. If a crash event follows an invocation event in a per-process history, we replace the crash event with an unsuccessful return event. If a crash event follows a return event, we simply discard the crash event. This way, we obtain a well-formed history, and can use the formalism in [7] to reason about histories. Moreover, our transformation preserves the semantics of histories: a partial write operation may or may not update the system, which is exactly the semantics of a write operation that returns NOK.

3.3 Specification of storage registers

We can now specify the properties of a storage register as the set of histories that may occur when n processes interact with it. Any history H in this set satisfies the following constraints:

CONSISTENCY**:** H is linearizable.

TERMINATION**:** For any process p, if $H|p$ contains an invocation event, then $H|p$ either contains a subsequent return event or a subsequent crash event.

PROGRESS**:** If only a single process p has a history $H|p$ that contains an infinite number of invocation events, and if p is correct, then $H|p$ contains an infinite number of successful return events.

The TERMINATION property states that operations should be non-blocking, that is, they should eventually return (unless the caller crashes). Notice that TERMINATION only insists that operations return, not that they return successfully. Without the PROGRESS property, an algorithm that simply returns NIL for every invocation of read, and NOK for every invocation of write, would be correct. The PROGRESS property precludes such trivial solutions.

4 Algorithm

4.1 Overview

In our system, read and write operations can be coordinated by any process. Each process provides register methods read() and write(val) that communicate asynchronously with other processes to coordinate an operation; similarly, each process provides message handlers that respond to requests from a coordinating process.

Each read and write operation contacts a majority of replicas. The term "quorum" denotes the set of replicas that participate in a particular operation.

Because a write quorum and a subsequent read quorum may be different, the latter set may contain replicas with different register values. To determine the current value, we associate a timestamp with values. This timestamp reflects the (logical) time at which the value was written. The basic task of a read

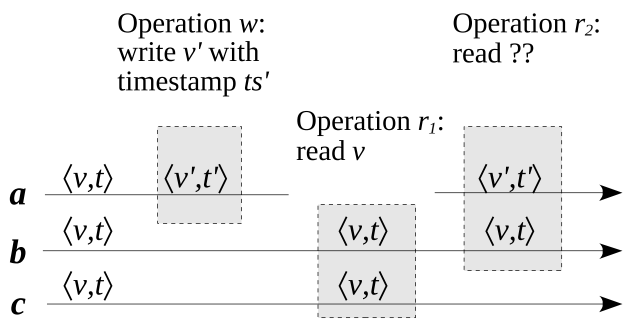

Figure 1: *Execution with a partial write operation with three replicas a, b, and c. Time flows to the right. $\langle v, t\rangle$ indicates that a process stores value v and timestamp t.*

operation is then to pick the most recent value in the read quorum, ensure that a majority stores this value, and return the value as the current register value. The basic task of a write operation is to store the new value in a quorum with a timestamp higher than any of the quorum's current timestamps.

A key complexity of the algorithm stems from the handling of a partial write operation, which stores a value in only a minority of replicas, either because the coordinator crashes or proposes too small a timestamp. After a partial write, a read operation cannot simply pick the value with the highest timestamp, since this may lead to a non-linearizable sequence of write and read operations. Figure 1 shows an example. Initially, all three replicas store value v with timestamp t. A write operation w, coordinated by a, stores value v' with timestamp t' ($t < t'$) only on a (e.g., because a crashes immediately afterward). A read operation r_1 then reads v from b and c. Finally, a recovers and runs a read operation r_2 on a and b. Consider now a history where w, r_1 and r_2 are causally connected (say, through the same external client) in the temporal order w, r_1, r_2. Since w fails, the register state is unknown after w; however, since r_1 returns v, the history indicates that w did not take effect. Thus, for consistency, r_2 must return v even though it finds the value v' with a higher timestamp.

One way to ensure linearizability in the presence of partial writes is to have every read operation update timestamps. For example, in Figure 1, by having r_1 update the timestamp in b and c, r_2 will see v having a timestamp greater than that of v'. However, we implement a more efficient scheme where read operations update the timestamps only when they detect partial writes. To allow this detection, we run a write operation in two phases. In the first phase, a write operation informs a majority about the intention to write a value; in the second phase, a write operation actually writes the value. A read operation can then detect a partial write as an unfulfilled intention to write a value.

We can now run read operations optimistically in a single round trip. A read operation first checks if a majority has the same timestamp and has seen no partial write; if so, it simply returns its own value. Failing the optimistic phase, the read operation picks the value in a majority that has the highest timestamp, and stores the chosen value in a majority with a timestamp that is greater than the timestamp of any previous write operation, including any partial write operations.

4.2 Detailed description

Each replica keeps three persistent variables: *val*, *val-ts*, and *ord-ts*, with initial values of $\perp$, $\mathsf{initialTS}$, and $\mathsf{initialTS}$, respectively. Variable *val* holds the current value of the register. Timestamp *val-ts* shows the logical time at which val was last updated. Timestamp *ord-ts* shows the logical time at which the most recent write operation was started, establishing its place in the ordering of operations. *val-ts* $<$ *ord-ts* indicates the presence of a partial write operation.

Algorithm 1 describes the register methods and Algorithm 2 the register message handlers. Methods "read" and "write" are public methods that can be invoked by any replica at any time. The internal procedure "majority" repeatedly sends a request to all replicas until the coordinator receives replies from a majority. We assume that there is a way for the coordinator to match requests and replies, e.g., by embedding a unique identifier in each request and reply.

The write method triggers a two-phase interaction with a majority. In the first phase, the coordinator sends "[Order,..]" messages to replicas with a newly generated timestamp. A replica updates its *ord-ts* and responds OK if it has not already seen a request with a higher timestamp. This establishes a place for the operation in the ordering of operations in the system and prevents a concurrent write operation with an older timestamp from storing a new value between the first

Algorithm 1 Register methods

```
1: procedure read()
2:     replies ← majority([Read, val-ts])
3:     if the status in all replies is true then return val
4:     else return recover()
5: procedure write(val)
6:     ts ← newTS()
7:     replies ← majority([Order, ts])
8:     if any status in a reply is false then return NOK
9:     replies ← majority([Write, val, ts])
10:    if the status in all replies is true then return OK
11:    else return NOK
12: procedure recover()
13:    ts ← newTS()
14:    replies ← majority([Order&Read, ts])
15:    if any status in a reply is false then return NIL
16:    val ← the value with highest val-ts from replies
17:    replies ← majority([Write, val, ts])
18:    if the status in all replies is true then return val
19:    else return NIL
20: procedure majority(msg)
21:    Send msg to all, retransmitting periodically
22:    await receive(rep) from ⌈(n+1)/2⌉ processes
           such that rep matches msg
23:    return set of received replies
```

Algorithm 2 Register message handlers

```
24: when receive [Read, ts] from coordinator
25:    status ← (ts = val-ts and ts ≥ ord-ts)
26:    reply [Read-R, status] to coordinator
27: when receive [Order, ts] from coordinator
28:    status ← (ts > max(val-ts, ord-ts))
29:    if status then ord-ts ← ts; store(ord-ts)
30:    reply [Order-R, status] to coordinator
31: when receive [Write, new-val, ts] from coordinator
32:    status ← (ts > val-ts and ts ≥ ord-ts)
33:    if status then
34:        val ← new-val; store(val)
35:        val-ts ← ts; store(val-ts)
36:    reply [Write-R, status] to coordinator
37: when receive [Order&Read, ts] from coordinator
38:    status ← (ts > max(val-ts, ord-ts))
39:    if status then ord-ts ← ts; store(ord-ts)
40:    reply [Order&Read-R, val-ts, val, status]
```

and second phases. In the second round, the coordinator sends "[Write,..]" messages and stores the value in a (possibly different) majority.

The read method first optimistically assumes that a majority of replicas have the same value and timestamp, and that there are no partial writes. If this assumption turns out to be true, the read method returns in line 3 after a single round-trip without modifying the persistent state of any replica. Otherwise, the two-phase recovery method is invoked, which works like the write method, except that it dynamically discovers the value to write. In the "[Order&Read...]" phase, a majority return their current *val-ts* and *val*. After this phase, the coordinator picks the value with the highest *val-ts* and writes it back to a majority using "[Write,...]" messages. This method ensures that, on recovery, the completed read operation appears to happen after the partial write operation and that future read operations will return values consistent with this history.

Our algorithm does not rely on the assumption that an update operation stores both val and the val-ts timestamp as a single atomic operation. Thus, it is correct even if a replica writes val to stable storage, but crashes before writing val-ts to stable storage. Recovery will detect and resolve any resulting disparity between timestamps at different replicas and return the stored value correctly.

5 Proof sketch

This section gives a sketch of a proof that our algorithm maintains linearizability, as defined in Section 3.2. The full proof of linearizability and liveness will appear in a separate technical report. *To reviewers: the full proof appears in the appendix of this paper.*

To simplify the presentation, we assume that each write tries to write a unique value and ignore the handling of the initial value $\perp$. In addition to the event histories defined in Section 3.2, we also consider internal *store events.* A store event happens when we store the variable *val* in stable storage in Line 34. A store event $\mathtt{st}(v, ts)$ corresponds to a store operation where *val* has value v and *val-ts* has value ts. For a history H, V_H is the set of values (a) successfully returned by a read request, or (b) for which a write method returned OK. For value $v \in V_H$, we define ts_v to be the smallest timestamp that is part of any store event for v.

We define a total ordering among V_H in the following way:

$$v < v' \Leftrightarrow ts_v < ts_{v'}. \qquad \text{(ORDER)}$$

We claim that our algorithm linearizes read and write requests in the order defined above. We prove our claim in two steps. Proposition 1 first proves that the order defined above "conforms" to history H. Proposition 4 shows that existence of a conforming total order is sufficient for a history to be linearizable.

Proposition 1 *Given a history H, the total order defined by* (ORDER) *satisfies the following properties.*

$$\mathsf{write}(v) \rightarrow_H \mathsf{write}(v') \wedge v, v' \in V_H \Rightarrow v < v' \quad (1)$$
$$\mathsf{read}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v' \quad (2)$$
$$\mathsf{write}(v) \rightarrow_H \mathsf{read}(v') \wedge v \in V_H \Rightarrow v \leq v' \quad (3)$$
$$\mathsf{read}(v) \rightarrow_H \mathsf{write}(v') \wedge v' \in V_H \Rightarrow v < v' \quad (4)$$

(Where $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2$ when the "return" of oper_1 happens before the "invocation" of oper_2 in H.)

Due to space constraints, we only prove the property (1) in this section. Proofs of other properties are similar. We first state several useful lemmas.

Lemma 2 *For any process p the values of val-ts and ord-ts increase monotonically.* (Proof is omitted.)

Lemma 3 *Assume that $\mathsf{write}(v) \in H$, and $v \in V_H$, and the coordinator of $\mathsf{write}(v)$ uses timestamp ts. Then, (a) some process executes $\mathsf{st}(v, ts)$, and (b) a majority has stored ts as their ord-ts at some moment during $\mathsf{write}(v)$.*

PROOF SKETCH: (a) is derived from the NO CREATION property of the channel. (b) is true, because the coordinator could not have sent "Write" message before a majority responding OK to "Order". □

PROOF SKETCH OF PROPERTY (1)

From Lemma 3, a majority maj_v has stored ts_v as their *ord-ts* at some moment. The same can be said for v' about a majority $maj_{v'}$ and $ts_{v'}$. Now, consider a process $p \in maj_v \cap maj_{v'}$. Because $\mathsf{write}(v) \rightarrow_H \mathsf{write}(v')$, p must have stored ts_v to the *ord-ts* before storing $ts_{v'}$. From Lemma 2, $ts_v < ts_{v'}$. □

Proposition 4 *Any history H is linearizable.*

PROOF SKETCH:We construct a sequential history S from H, such that (a) S contains all the events in H, and (b) for values $v < v'$ in H, S orders all events that involve v before events that involve v'. S is clearly equivalent to H in the sense defined in [7]. It then suffices to show that S is legal, i.e., a read operation in S returns the latest value stored in the register.

Assume that $\mathsf{read}(v)$ is in S. From (4), we can conclude that $\mathsf{write}(v) \rightarrow_S \mathsf{read}(v)$. We now have to show that there is no operation $\mathsf{write}(v')$ between $\mathsf{write}(v)$ and $\mathsf{read}(v)$ in S. Assume for a contradiction that such $\mathsf{write}(v')$ exists. From (1), we know that $v < v'$. At the same time, we know from (3) that $v' \leq v$, which is impossible. □

6 Practical considerations

6.1 Generating timestamps

So far, we have assumed that a node will eventually generate sufficiently large timestamps to ensure the system's theoretical safety and liveness. In practice however, the timestamps generated by nodes should be synchronized in order to enable quick progress. We use loosely synchronized real-time clocks for this purpose.

We have analyzed many traces, from a variety of real-world systems and applications, in order to determine the I/O behaviors exhibited. In particular, we are interested in the time differentials between writes to the same block of storage, as this determines the tightness of time synchronization required. Table 1 shows the properties of some of these workloads.

Cello: A file system managed by an 8 processor HP9000 N4000 for 20–30 researchers, with 16 GB of RAM, and an HP XP512 disk array.

SAP: SAP ISUCCS and Oracle supporting 3000 users and several background batch jobs running on an HP V2500 with an HP XP512 disk array.

OpenMail: An HP9000 K580 server with 6 CPUs, 3.75 GB of RAM, and an EMC 3700 Symmetrix disk array. Approximately 2000 users access their email during the course of the trace.

As even the smallest time differentials observed are on the order of several hundred microseconds, and modern time synchronization protocols, such as NTP, can usually synchronize clocks on the order of 10us [14], we believe that the system will handle the vast majority of requests without rejection.

6.2 Reducing the size of control information

Since we target storage systems with large capacities (up to peta bytes of storage capacity), the amount of control information per register is an important complexity measure.

We can reduce our control information from two persistent timestamps per register to one. The *ord-ts* is only needed to capture information about in-progress update operations. Once *ord-ts* and *val-ts* are equal, we know that the place for this operation, reserved in the ordering of events by the *ord-ts*, has been confirmed by the updating of the *val-ts*. At any point in time, we expect the number of in-progress update operations to be dramatically smaller than the total number of blocks in the system. Thus, the control information embodied by the *ord-ts* timestamp can be stored in a dynamic log. Because there is nothing inherently complicated about using a log instead of a second timestamp, the second timestamp *ord-ts* is used in this paper to simplify presentation of our algorithm.

We can further reduce our control information to eliminate timestamps in the case where no replicas have failed. The *val-ts* is needed to capture information about locally applied update operations. Once the *val-ts* is the same at *all* replicas, we know that the update operation has succeeded at all replicas. Thus, the coordinator can run an extra phase on completion of an update operation where, if *all* replicas have reported successful completion of the write, the replicas may remove *val-ts*.

6.3 Algorithm complexity

Table 2 compares the performance of our algorithm and state-of-the-art atomic-register constructions that perform two-round messaging for both reading and writing [11, 12]. We improve the previous work especially in the common case of reading from a register in the absence of failures or concurrent accesses.

7 Conclusions and future work

We have described a new replication protocol suitable for logical disk systems. The main contributions of the paper are the following:

- The extension to linearizability to model crash-recoverable data objects.
- The specification of a storage register that reflects the properties of logical disk systems.
- An implementation of this register that is more efficient than existing atomic-register constructions.

We have implemented a prototype of this protocol on a cluster of PCs. (we refer to the whole as a Federated Array of Bricks, or FAB). We are currently studying the systems behavior and performance under various situations, including failures and overloads.

We have identified two major areas of future work. One is dynamic volume reconfiguration after failures or to improve performance. We plan to adapt the technique of [12], by superimposing a new quorum configuration asynchronously, transferring contents to new bricks, and garbage collecting old quorum configurations in the background.

The other is reducing the storage overhead of quorum-based replication using witnesses and witness promotion. We adapt the timestamp-discarding scheme introduced in Section 6.2 to create "witness" replicas that only keep timestamps, but no actual block values (at least in the long term). By replicating a logical segment on only $f + 1$ normal replicas and f additional witnesses, the segment can tolerate f failures with little space overhead.

References

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

workload	*date*	*length*	*total data*	*#writes*	*#reads*	*smallest write delay*	*99% write delay*
Cello	9/2002	1 day	1.4 TB	5,250,126	6,766,002	0.26 ms	5.9 ms
SAP	1/2002	15 min	5 TB	150,339	4,835,793	2.0 ms	5.6 ms
OpenMail	10/1999	1 hr	7 TB	931,979	355,962	0.47 ms	2.9 ms

Table 1: *Workload characteristics.* Date *shows when the trace was collected. Smallest write delay is the smallest time between write requests to the same data block. 99% write delay shows the 99% boundary for write delays (i.e. only 1% of the writes will have a delay smaller than this time).*

	Our algorithm			LS97	
	fast read	slow read	write	read	write
latency	2δ	6δ	4δ	4δ	4δ
# messages	$2n$	$6n$	$4n$	$4n$	$4n$
# disk reads	1	$n+1$	0	n	0
# disk writes	0	n	n	n	n
Network bandwidth consumption	B	$(2n+1)B$	nB	$2nB$	nB

Table 2: *Performance comparison between our algorithm and the one by Lynch and Shvartsman [11]. "Fast read" refers to the optimistic part of a read method. "Slow read" refers to a read method that executes recovery. δ is the maximum one-way messaging delay. n is the number of replicas—we pessimistically assume that all replicas are involved in the execution of an operation. When calculating the number of disk I/Os, we assume that reading val and* store(*val*) *involve single disk read and write, whereas ts and ord-ts are stored on NVRAM (or volatile memory backed up with a dedicated transactional log device). B is the size of a register.*

[2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[3] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *International Symposium on Distributed Computing (DISC)*, 2002.

[4] M. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[5] E. Gafni and L. Lamport. Disk paxos. In *International Symposium on Distributed Computing (DISC)*, 2000.

[6] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th. Symposium on Operating Systems Principles*, 1979.

[7] 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.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

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

[10] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[11] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the IEEE Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 272–281, 1997.

[12] N. A. Lynch and A. A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, October 2002.

[13] D. Malkhi and G. Chockler. Active disk paxos with infinitely many processes. In *Proceedings of the ACM conference on Principles of Distributed Computing (PODC)*, 2002.

[14] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, United Kingdom, September 1994.

A Correctness

A.1 Histories

In accordance with the model in Section 3, we use a history of invocation and return events to represent the interaction between processes and an instance of our register during a particular run of our algorithm. Since there are no partial operations in our histories (we replace crash events with unsuccessful return events as necessary), our histories are well-formed and complete according to the definition in [7].

Rather than refer to individual invocation and return events in a history, we use a notion of operation, which is an aggregation of an invocation event and a return event. Where events are totally ordered, operations are only partially ordered (they may overlap). If the return event of an operation oper precedes the invocation event of another operation oper' in a history H, we say that oper happens before oper' and we write this as $\mathsf{oper} \rightarrow_H \mathsf{oper}'$.

Each operation contains a value from the set Value. We use $\mathsf{write}(v)$ to represent a write operation that writes the value v. We use $\mathsf{read}(v)$ to represent a (successful) read operation that returns v. To simplify the presentation, we assume that each invocation of the write operation tries to write a unique value (*"unique-value" assumption*). The value $\perp$ ($\perp \in \mathsf{Value}$) represents the initial value of the register. We assume that $\perp$ is not part of any write operation (if $\mathsf{write}(v)$ is in H, then $v \neq \perp$).

For an operation oper, we denote $\mathsf{coord}(\mathsf{oper})$ to represent the process that coordinates oper, and $\mathsf{ts}(\mathsf{oper})$ to represent the timestamp used by $\mathsf{coord}(\mathsf{oper})$ (for a read operation, coord is defined only when the recover method is executed.). Notations $\mathsf{maj}_R(\mathsf{oper})$, $\mathsf{maj}_W(\mathsf{oper})$, $\mathsf{maj}_O(\mathsf{oper})$ represent a majority of processes contacted by the coordinator after successful completion of "Read", "Write", and "Order"/"Order&Read" phases, respectively.

For a given history H, we define the following subsets of Value:

- $\mathsf{Written}_H$ is the set of all values that are part of invocation events for write operations in H.
- $\mathsf{Commited}_H$ is the set of all values that are part of an invocation event for a write operation that return a status of OK in H.
- Read_H is the set of all values that are part of a return event for a read operation in H.

We also call the set $\mathsf{Read}_H \cup \mathsf{Commited}_H$ the *observable* values in H, and define

$$\mathsf{Obs}_H \equiv \mathsf{Read}_H \cup \mathsf{Commited}_H.$$

A.2 Internal events

Histories represent the external view of a register: they reflect the view of processes that interact with the register. We also consider internal events that happen during a run of our algorithm. We use these internal events to reason about the behavior of our algorithm, and to justify that the algorithm implements an external behavior that complies with the specification in Section 3.

A *store event* is an internal event that corresponds to the invocation of store, triggered by the handler for "Write" messages in Line 31. We use $\mathsf{st}(v, ts)$ to denote a store event that writes a value v in the context of timestamp ts.

We use SE_R to denote the set of store events that happen in a run R. SE_R^v is the (possibly empty) set of store events for value v. If $\mathsf{SE}_R^v \neq \emptyset$, we use ts_v to denote the smallest timestamp that is part of any store event in SE_R^v.[3]

Lemma 5 *ts_v, if it exists, is the timestamp used by the write method to send "Write" messages to all processes.*

PROOF: A store event $\mathsf{st}(v, ts_v)$ may happen in two cases:

(a) An operation $\mathsf{write}(v)$ with timestamp ts_v issues the "Write" message (Line 9). This case proves our claim.

(b) An operation $\mathsf{read}(v)$ executes a recover method. It finds a value v from some reply (from, say, process p) during the "Order&Read" phase and sends "Write" afterward. This case is impossible for the following reason. Process p must have executed $\mathsf{st}(v, ts')$ for some timestamp ts'. Moreover, $ts' < \mathsf{ts}(\mathsf{read}(v)) = ts_v$ from Line 29. This

[3] Although ts_v is defined for a particular run, we do not parameterize ts_v with that run for brevity.

contradicts our assumption that ts_v is the smallest timestamp among store events involving v. □

Lemma 6 *If a process executes* $\mathsf{st}(v, ts)$ *for some value* v *and timestamp* ts*, then a majority has stored* ts *as the value of ord-ts.*

PROOF: Event $\mathsf{st}(v, ts)$ happens only after the coordinator collected either "Order-R" or "Order&Read-R" replies from a majority. The handler for the messages "Order" or "Order&Read" set *ord-ts* to ts. □

For a run R, we define SV_R to be the set of values that are part of store events in R. Notice that $\perp \notin \mathsf{SV}_R$ because $\perp$ is never written.

Lemma 7 *For any run* R *that gives rise to a history* H*,*

$$\mathsf{Obs}_H \setminus \{\perp\} \subseteq \mathsf{SV}_R \subseteq \mathsf{Written}_H.$$

PROOF: From NO CREATION property of the communication channels, and Algorithms 1 and 2. □

For any run R, we can define a total order $<_{val}$ on $\mathsf{SV}_R \cup \{\perp\}$ in the following manner:

$$\perp <_{val} v \qquad v \in \mathsf{SV}_R \tag{5}$$
$$v <_{val} v' \quad \Leftrightarrow ts_v < ts_{v'} \quad v, v' \in \mathsf{SV}_R \tag{6}$$

This is a well-defined total order because, from the unique-value assumption, different values are always stored with different timestamps ($v \neq v' \Rightarrow ts_v \neq ts_{v'}$). In the following, we omit the subscript from $<_{val}$, and simply use "$<$". With this convention, the symbol $<$ is overloaded to order both timestamps and values.

As we show subsequently, our algorithm linearizes operations in accordance with this total order.

A.3 Proof of safety

We first define the notion of a *conforming total order* for a history. Intuitively, a conforming total order for a history H is a totally-ordered set $(V, <)$ such that (a) V contains all the obervable values in H, and (b) the ordering of values in V corresponds to the ordering of operations in H.

Definition 8 *Given a history* H*. A totally ordered set* $(V, <)$ *is a conforming total order for* H *if the following conditions are satisfied:*

$$\mathsf{Obs}_H \subseteq V \subseteq \mathsf{Written}_H \cup \{\perp\} \tag{7}$$
$$\mathsf{write}(v) \to_H \mathsf{write}(v') \wedge v, v' \in V \Rightarrow v < v' \tag{8}$$
$$\mathsf{read}(v) \to_H \mathsf{read}(v') \Rightarrow v \leq v' \tag{9}$$
$$\mathsf{write}(v) \to_H \mathsf{read}(v') \wedge v \in V \Rightarrow v \leq v' \tag{10}$$
$$\mathsf{read}(v) \to_H \mathsf{write}(v') \wedge v' \in V \Rightarrow v < v' \tag{11}$$

□

Using the concept of conforming total order, our safety proof proceeds in two steps. In Section A.3.1, we prove that, for any run R that results in a history H, the set of stored values SV_R with the total order defined in (5) and (6) is a conforming total order for H. Then, in Section A.3.2, we show that the existence of a conforming total order for a history H is a sufficient condition for H being linearizable.

A.3.1 Proving the existence of a conforming total order

This section proves that the total order we defined in (6) is, in fact, a conforming total order for any history H.

Lemma 9 *For any processor* p*, the value of val-ts and ord-ts increases monotonically in any history.*

PROOF: Variable *val-ts* is modified only at Line 35, which checks beforehand if the new timestamp is larger than the current one. Variable *ord-ts* is modified only in Line 29, which also checks beforehand if the new timestamp is larger than the current one. □

Lemma 10 *Given two distinct values* $v, v' \in \mathsf{SV}_R$*. If there exists a timestamp* $ts > ts_{v'}$ *such that a majority of processes execute* $\mathsf{st}(v, ts)$*, then every store event for* v' *has a timestamp smaller than* ts*:* $\forall\, \mathsf{st}(v', ts') \in \mathsf{SE}_R^{v'} : ts' < ts$.

PROOF: Assume the contrary. Let ts'_{min} be the smallest timestamp for store events involving v':

$$ts'_{min} \equiv \min(\{ts' : ts' > ts \wedge \mathsf{st}(v', ts') \in \mathsf{SE}_R^{v'}\}).$$

We first argue that the event $\mathsf{st}(v', ts'_{min})$ must be triggered by a recover method. This is because, from Lemma 5, only the original write method for v' uses $ts_{v'}$ when storing v'. Thus, any store event $\mathsf{st}(v', ts')$ with $ts' > ts_{v'}$, must be executed as a part of a recover method.

Consider now the recover method that triggers $\mathsf{st}(v', ts'_{min})$. Let ts'' be the highest timestamp returned in a "Order&Read-R" message that is received as part of this recover method invocation (Line 16). Notice that $ts'' \geq ts$ for the following reasons.

- Consider a process $p \in \mathsf{maj}_O(\mathsf{st}(v', ts'_{min})) \cap \mathsf{maj}_W(\mathsf{st}(v, ts))$ Process p cannot send "Order&Read-R" for v' before $\mathsf{st}(v, ts)$ for the following reason: when sending "Order&Read-R", p's *ord-ts* = $ts'_{min} > ts$. Executing $\mathsf{st}(v, ts)$ afterwards violates Lemma 9,
- Thus, p must execute $\mathsf{st}(v, ts)$ before "Order&Read-R". This means that, upon "Order&Read-R", p must return a timestamp, say ts_o, such that $ts_o > ts$. On the other hand, by definition, $ts'' \geq ts_o$.
- Thus, $ts'' \geq ts$.

Moreover, because the recover method triggers the writing of v', v' must have timestamp ts'' at some process. Since $v \neq v'$, we can now conclude that this process executed a store event $\mathsf{st}(v', ts''')$ with $ts \leq ts'' < ts'''$. This contradicts the assumption that ts'_{min} is the smallest timestamp bigger than ts for which v' is stored. □

Lemma 11 *If a run R gives rise to a history H, and if* $\mathsf{read}(v) \in H$ *with* $v \neq \perp$*, then there exists a timestamp ts such that (a) a majority executes* $\mathsf{st}(v, ts)$ *and (b) a majority has ts as their persistent val-ts timestamp sometime during the execution of* $\mathsf{read}(v)$.

PROOF: Assume that H contains $\mathsf{read}(v)$. There are two ways in which $\mathsf{read}(v)$ can be executed:

- $\mathsf{read}(v)$ only involves the invocation of a read method. In this case, processes in $\mathsf{maj}_R(\mathsf{read}(v))$ must have returned the same timestamp ts with no pending write or recover invocations (Line 3). Thus, a majority executed $\mathsf{st}(v, ts)$, and a majority has ts as their *val-ts* during $\mathsf{read}(v)$.
- $\mathsf{read}(v)$ involves the read and recover operations. Let $ts = \mathsf{ts}(\mathsf{read}(v))$. For the recovery to succeed, $\mathsf{maj}_W(\mathsf{read}(v))$ must have executed $\mathsf{st}(v, ts)$ and replied "Write-R" to the coordinator, in which case a majority has ts as their *val-ts* timestamp. □

Lemma 12 *Given a run R that gives rise to a history H. If* $\mathsf{write}(v)$ *is in H and* $v \in \mathsf{Obs}_H$*, then (a) some process executes* $\mathsf{st}(v, ts_v)$*, (b) there is a timestamp ts such that a majority executes* $\mathsf{st}(v, ts)$.

PROOF: If $v \in \mathsf{Commited}_H$, then all the properties hold vacuously. Suppose otherwise.

(a) Because $\mathsf{read}(v)$ is in H, some process must have stored v to its *val* via a store event $\mathsf{st}(v, ts)$ for some timestamp ts (ts_v is merely the smallest among such timestamps).

(b) From Lemma 11, a majority executes $\mathsf{st}(v, ts)$ for some ts. □

Lemma 13 *Given a run R that gives rise to a history H. If* $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2$ *and both operation trigger some store events, then the store events for* oper_1 *have smaller timestamps than those for* oper_2.

PROOF: Assume the contrary: $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2$, oper_1 executes $\mathsf{st}(v, ts_1)$, oper_2 executes $\mathsf{st}(v', ts_2)$, yet $ts_1 > ts_2$.

Since oper_1 executes $\mathsf{st}(v, ts)$, processes in $\mathsf{maj}_O(\mathsf{oper}_1)$ store ts_1 for *ord-ts* at some point in the history (happens in Line 29 or Line 39). Similarly, processes in $\mathsf{maj}_O(\mathsf{oper}_2)$ store ts_2 as their *ord-ts* at some point in the history. Consider a process $p \in \mathsf{maj}_O(\mathsf{oper}_1) \cap \mathsf{maj}_O(\mathsf{oper}_2)$. Since $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2$, this process stores ts to *ord-ts* before it stores ts' to *ord-ts*. Since processes only assign monotonically increasing values to *ord-ts* (Lemma 9), we have a contradiction. □

Lemma 14 *For any run R that gives rise to a history H, the condition holds:*

$$\mathsf{write}(v) \rightarrow_H \mathsf{write}(v') \wedge v, v' \in \mathsf{Obs}_H \Rightarrow v < v'$$

PROOF: Assume otherwise: $\mathsf{write}(v) \rightarrow_H \mathsf{write}(v')$, $v, v' \in \mathsf{Obs}$, yet $v > v'$. From Lemma 12, we know that $\mathsf{st}(v, ts_v)$ happens during $\mathsf{write}(v)$ and that $\mathsf{st}(v', ts_{v'})$ happens during $\mathsf{write}(v')$. From Lemma 13, we conclude that $ts_v < ts_{v'}$. □

Lemma 15 *For any run R that gives rise to a history H, the following condition holds:*

$$\mathsf{read}(v) \rightarrow_H \mathsf{read}(v') \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{read}(v) \rightarrow_H \mathsf{read}(v')$, yet $v > v'$. This means that $v \neq \perp$. From Lemma 11, for some timestamp ts, either $\mathsf{maj}_R(\mathsf{read}(v))$ or $\mathsf{maj}_W(\mathsf{read}(v))$ has ts as their value for *val-ts* some time during the execution of $\mathsf{read}(v)$. Let maj_v be this majority set.

Consider first the case where $v' = \perp$. Observe first that $\mathsf{read}(\perp)$ can execute only read(). For a read method to return $\perp$, $\mathsf{maj}_R(\mathsf{read}(v'))$ has $\mathsf{initialTS}$ as their value for *val-ts* and *ord-ts*. Let $p \in maj_v \cap \mathsf{maj}_R(\mathsf{read}(v'))$. Because p has increasing values for its *ord-ts* timestamp, and because $\mathsf{read}(v)$ precedes $\mathsf{read}(\perp)$, we have $ts < \mathsf{initialTS}$, which is a contradiction.

Consider next the case where $v' \neq \perp$. From Lemma 11, for some timestamp ts', a majority executes $\mathsf{st}(v', ts')$ and either $\mathsf{maj}_R(\mathsf{read}(v'))$ or $\mathsf{maj}_W(\mathsf{read}(v'))$ has ts' as their value for *val-ts* sometime during the execution of $\mathsf{read}(v')$. Let maj'_v be this majority set. Let $p \in maj_v \cap maj'_v$. Because $\mathsf{read}(v)$ precedes $\mathsf{read}(v')$, and from Lemma 9, we conclude that $ts < ts'$.

Moreover, $ts_{v'} < ts_v < ts < ts'$. ($ts_{v'} < ts_v$ because $v' < v$; $ts_v < ts$ from the definition of ts_v). Since maj_v executes $\mathsf{st}(v, ts)$, Lemma 10 implies that all store events for v' have a timestamp that is smaller than ts. But this contradicts the fact that maj'_v executes $\mathsf{st}(v', ts')$ with $ts < ts'$. □

Lemma 16 *For any run R that gives rise to a history H, the following condition holds:*

$$\mathsf{write}(v) \rightarrow_H \mathsf{read}(v') \wedge v \in \mathsf{Obs}_H \Rightarrow v \leq v'$$

PROOF: Assume for a contradiction that $\mathsf{write}(v) \rightarrow_H \mathsf{read}(v')$, $v \in \mathsf{Obs}_H$, yet $v > v'$.

We first show that there exists a timestamp $ts' > ts_v$ such that a majority executes $\mathsf{st}(v', ts')$. We consider two situations:

(a) $\mathsf{read}(v')$ executes the recover method .
Let $ts' = \mathsf{ts}(\mathsf{read}(v'))$. We know that a majority executes $\mathsf{st}(v', ts')$. Furthermore, from Lemma 12, we know that at least one store event $\mathsf{st}(v, ts_v)$ happens during $\mathsf{write}(v)$. From Lemma 13, $ts_v < ts'$.

(b) $\mathsf{read}(v')$ executes only the read method.
Then $\mathsf{maj}_R(\mathsf{read}(v'))$ has some timestamp ts as their value for both *ord-ts* and *val-ts*. According to Lemmas 6 and 12, $\mathsf{maj}_O(\mathsf{write}(v))$ has ts_v as their value for *ord-ts* somtime during $\mathsf{write}(v)$. Consider a process $p \in \mathsf{maj}_R(\mathsf{read}(v')) \cap \mathsf{maj}_O(\mathsf{write}(v))$. From Lemma 9 and the fact that $v \neq v'$, $ts_v < ts$. In particular, we then know that $ts \neq \mathsf{initialTS}$, and therefore that $v' \neq \perp$. Since $v' \neq \perp$, we conclude that $\mathsf{maj}_R(\mathsf{read}(v'))$ executed $\mathsf{st}(v', ts)$.
Let ts be a timestamp such that a majority executes $\mathsf{st}(v, ts)$—Lemma 12 guarantees the existence of ts. Per definition, we know that $ts_v < ts$. From the above reasoning, we also have a timestamp ts' such that a majority executes $\mathsf{st}(v', ts')$ and such that $ts_v < ts'$.

We now have one of two situations: (c) $ts > ts'$ or (d) $ts < ts'$. For (c), we have $ts_v < ts' < ts$, which contradicts Lemma 10. For (d), we have $ts_{v'} < ts_v < ts'$, which also contradicts Lemma 10. □

Lemma 17 *For any run R that gives rise to a history H, the following condition holds:*

$$\mathsf{read}(v) \rightarrow_H \mathsf{write}(v') \wedge v' \in \mathsf{Obs}_H \Rightarrow v < v'$$

PROOF: Assume for a contradiction that $\mathsf{read}(v) \rightarrow_H \mathsf{write}(v')$, $v' \in \mathsf{Obs}_H$, yet $v \geq v'$. We know that $v' \neq \perp$, and can therefore conclude that $v \neq \perp$. There are two cases to consider regarding $\mathsf{read}(v)$:

(a) $\mathsf{read}(v)$ executes only the read method.
We know that $\mathsf{maj}_R(\mathsf{read}(v))$ has some timestamp ts their *ord-ts* and *val-ts* timestamps during $\mathsf{read}(v)$. From Lemmas 6 and 12,

$\mathsf{maj}_O(\mathsf{write}(v'))$ has $ts_{v'}$ as their value for *ord-ts* during $\mathsf{write}(v')$. Consider a process $p \in \mathsf{maj}_R(\mathsf{read}(v)) \cap \mathsf{maj}_O(\mathsf{write}(v'))$. From Lemma 9, we know that $ts < ts_{v'}$. Since a majority executes $\mathsf{st}(v, ts)$, $ts_v < ts$. Thus, we conclude that $ts_v < ts < ts_{v'}$, which contradicts the assumption that $v > v'$.

(b) $\mathsf{read}(v)$ executes both the read and recover methods.

Let ts be the timestamp used in the recover method. From Lemma 12, a store event $\mathsf{st}(v', ts_{v'})$ happens during $\mathsf{write}(v')$. From Lemma 13, we know that $ts < ts_{v'}$. As for case (a), we can now derive a contradiction based on the fact that $ts_v < ts$. □

A.3.2 Proof of linearizability

Proposition 18 *Given a history H. If there exists a conforming total order for H, then H is linearizable.*

PROOF: Let $(V, <)$ be a conforming total order for H. Construct a sequential history S that has the same events as H and that satisfies the following conditions:

1. $(V, <)$ is a conforming total order for S.
2. $\mathsf{oper}_1 \rightarrow_H \mathsf{oper}_2 \Rightarrow \mathsf{oper}_1 \rightarrow_S \mathsf{oper}_2$.

It is possible to satisfy (1) by simply ordering the operations in S such that if $v < v'$, all operations for v precede all operations for v'. Moreover, we can satisfy (1) without violating (2). Consider two operations oper_1 and oper_2 in H. If these operations are concurrent, we can order them in any way without violating (2). If $oper_1 \rightarrow_H \mathsf{oper}_2$, their ordering in H already obeys the ordering of values in V. This is because $(V, <)$ is a conforming total order for H.

Condition (2) implies that S and H are equivalent (according to the definition of equivalence in [7]), and that the ordering of H is a subset of the ordering in S. Thus, to prove that H is linearizable, it is now sufficient to show that S is legal (i.e., that S is in the sequential specification of our register). To show that S is legal, we have to show that all read operations in S either (a) return the latest value stored in the register or (b) return $\perp$ if no value has been stored in the register.

Consider first case (b). We have to show that if $\mathsf{write}(v) \rightarrow_S \mathsf{read}(\perp)$ then $v \notin \mathsf{Obs}_S$. Assume for a contradiction that $\mathsf{write}(v) \rightarrow_S \mathsf{read}(\perp)$ with $v \in \mathsf{Obs}_S$. We can use the fact that $(V, <)$ is a conforming total order. From 10, we know that $v \leq \perp$. But since $\perp$ is not written, we can conclude that $v < \perp$, which is a contradiction with (5).

Consider next (a). Assume that $\mathsf{read}(v) \in S$ with $v \neq \perp$. Since v is in Read_S, we know that $v \in V$. From (7), we can derive that $\mathsf{write}(v) \in S$. Moreover, from (11), we can conclude that $\mathsf{write}(v) \rightarrow_S \mathsf{read}(v)$. We now have to show that there is no operation $\mathsf{write}(v')$ between $\mathsf{write}(v)$ and $\mathsf{read}(v)$ in S. Assume for a contradiction that $\mathsf{write}(v) \rightarrow_S \mathsf{write}(v')$ and that $\mathsf{write}(v') \rightarrow_S \mathsf{read}(v)$. From (8), we know that $v < v'$. At the same time, we know from (10) that $v' \leq v$, which is impossible.

□

Proposition 19 (CONSISTENCY) *Any run produces a linearizable history.*

PROOF: Given a history H. We first prove that the set Obs_H with the total order $<$ defined in (6) is a conforming total order for H.

Lemma 7 proves (7). Lemma 14 proves (8), Lemma 15 proves (9), Lemma 16 proves (10), and Lemma 17 proves (11). We can now conclude that $(\mathsf{Obs}_H, <)$ is a conforming total order for H.

The linearizability of H then follows from Proposition 18. □

A.4 Proof of liveness

Lemma 20 *If a process invokes the "majority" procedure in Algorithm 1, and then does not crash, the invocation eventually returns.*

PROOF: Assume that a process p invokes "majority" with a message m, and then does not crash. Assume furthermore for a contradiction that the invocation of majority does not return, i.e., that "majority" will send m an infinite number of times. From the FAIR LOSS property of the channel, all correct processes receive m an infinte number of times. Because there is a majority of correct processes, we know there is a time

t after which a majority of processes does not crash. When this majority receives m after t, each process in the majority will send a reply to p. Because each process in the majority receives m an infinite number of times, they will each send an infinite number of replies. Again, by the FAIR LOSS property, p will receive an infinite number of replies from a majority. Thus, the **await** statement in Line 22 will eventually return, leading to a contradiction. $\square$

Proposition 21 (TERMINATION) *For any process p, if $H|p$ contains an invocation event, then $H|p$ either contains a subsequent return event or a subsequent crash event.*

PROOF: Assume for a contradiction that a process history $H|p$ contains an invocation event, but no subsequent return nor crash event.

Consider first the case where p invokes the read method. Because p does not crash after invoking read, we know from Lemma 20 that all invocations of majority during the read operation will eventually return. Moreover, we know that invocations of newTS are non-blocking. We conclude that the invocation of read will eventually return, which is a contradiction. We can derive a similar contradiction for invocations of write. $\square$

Proposition 22 (PROGRESS) *If only a single process p has a history $H|p$ that contains an infinite number of invocation events, and if p is correct, then $H|p$ contains an infinite number of successful return events.*

PROOF: Because p is the only process with an infinite number of invocation events, all other processes generate only a finite number of timestamps. Let ts be the maximum timestamp generated by processes other than p.

Assume that $H|p$ contains an infinite number of unsuccessful return events. From Algorithm 1, we can observe that each invocation with an unsuccessful return event causes the generation of a timestamp. Thus, we know that p generates an infinite number of timestamps. The PROGRESS property of timestamp ensures that p eventually generates a timestamp ts' that is higher than ts. Because p is correct, there is a time t such that (a) p does not crash after t and (b) p invokes a method after t and generates a timestamp ts_c that is greater than ts. Consider this invocation. No replica will reply NO during this invocation because ts_c is higher than any timestamp in the system. This means that the invocation will return successfully, which is a contradiction. $\square$