Decentralized Storage Consistency via Versioning Servers

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter

September 2002

CMU-CS-02-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes a consistency protocol that exploits versioning storage-nodes. The protocol provides linearizability with the possibility of read aborts in an asynchronous system that may suffer client and storage-node crash failures. The protocol supports both replication and erasure coding (which precludes post hoc repair of partial-writes), and avoids the excess work of two-phase commits. Versioning storage-nodes allow the protocol to avoid excess communication in the common case of no write sharing and no failures of writing clients.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by DARPA/ITO's OASIS program, under Air Force contract number F30602-99-2-0539-AFRL. Garth Goodson was supported by an IBM Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Keywords: Decentralized storage, consistency protocol, versioning servers, distributed file systems

1 Introduction

Survivable storage systems (e.g., Petal [19], Myriad [7], SwiftRAID [21], PASIS [35], and Cheops [3]) preserve and provide access to data even when a subset of storage-nodes fail. The common architecture for such systems spreads data redundantly (either via replication or erasure coding) across a set of decentralized storage nodes. Even when some have failed, the remaining storage-nodes can provide sufficient information to reconstruct stored data.

One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. That is, the set of fragments used by a reader must correspond to the same update operation. Such consistency is made difficult by concurrent updates from distinct clients, partial updates made by clients that fail, and failures of storage-nodes. Without such consistency, data loss is possible with replication and likely with erasure coding, since no storage-node has all of the data and mismatched fragments produce garbage. Protocols exist for achieving such consistency, but they generally involve significant extra work even in the common cases.

This paper describes a new approach to achieving such consistency, built upon internal data versioning within storage-nodes. If each storage-node keeps all versions of its fragments, it becomes possible to achieve useful consistency semantics without excess communication or I/O in the common cases of non-failed clients and minimal write sharing. Such versioning is increasingly common in storage systems because it simplifies several aspects of data protection, including recovery from user mistakes [32], recovery from system failure [14], and survival of client compromises [34]. Further, it can be implemented with minimal performance cost ($\approx 2\%$) [34] and capacity demand ($\approx 10\%$) [33].

The consistency semantic for which we strive is a variation of linearizability [13] that permits read operations to abort with no return value (rarely in practice). These semantics have been studied previously by Pierce, for example, under the name "pseudo-atomicity" [28]. Our contribution consists of a simple and practical protocol that exploits versioning to implement these semantics for decentralized, redundant storage. It elegantly handles partial writes without requiring two-phase commit or repair, which is particularly difficult for erasure coding schemes.

Briefly, the protocol works as follows. After polling storage-nodes to discover a logical time value, clients update a data-item by writing fragments to at least a write threshold of storage-nodes. Clients read a data-item by fetching the latest versions of fragments from a read threshold of storage-nodes; additional fragments and earlier versions are read, as necessary, until a consistent answer is found. A proof of the protocol's correctness is sketched, and optimizations are discussed. Then, the protocol is refined to replace logical time with loosely synchronized clocks, eliminating the extra round-trip for writers.

Our goal for this protocol is practical efficiency for survivable storage, and it is therefore tuned for common scenarios. In particular, most studies of distributed storage systems (e.g., [4, 10, 15, 26]) indicate that minimal writer-writer and writer-reader sharing occurs, usually well under 1% of operations. In the absence of such sharing, only the uncommon case of partial writes by failing clients requires more than the minimum work for consistency — in a sense, the system is extremely optimistic in that the system only involves extra work if a reader actually observes a concurrent write operation.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the system model assumed. Section 4 presents and proves our protocol. Section 5 extends the protocol to work with loosely synchronized client clocks, eliminating the extra roundtrip for acquiring logical time during a write operation. Section 6 discusses performance, Byzantine clients, and pruning of old versions.

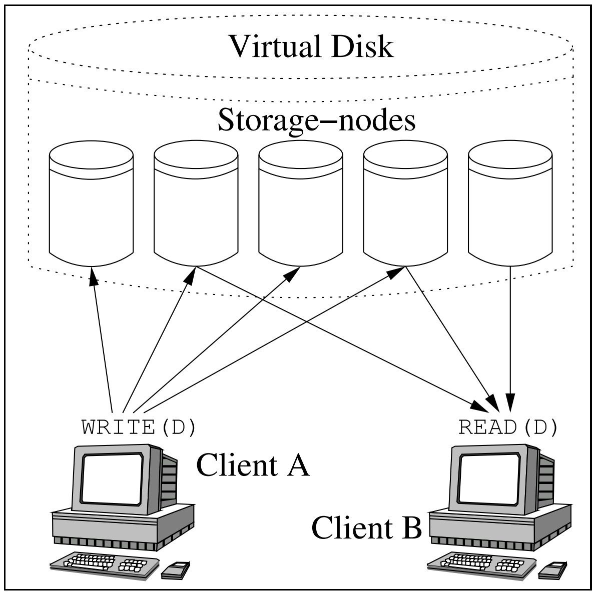

Figure 1: **High-level architecture for survivable storage.** Spreading redundant data across independent storage-nodes increases the likelihood of data surviving failures of storage-nodes. Clients update multiple servers to complete a write and (usually) readers fetch information from multiple servers to complete a read.

2 Background

Figure 1 illustrates the basic architecture of a fault-tolerant, or survivable, distributed storage system. To write a data-item D, Client A issues write requests to multiple storage-nodes, storing inter-related fragments. To read D, Client B issues read requests to an overlapping subset of storage-nodes. This basic scheme allows readers and writers to successfully access data-items even when subsets of the storage-nodes have failed. To provide reasonable storage semantics, however, the system must guarantee that readers see consistent answers—for example, assuming no intervening writes, two successful reads of D should produce the same answer (the most recent write) independent of which subset of storage-nodes are contacted.

A common data distribution scheme used in such storage systems is data replication. That is, a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. It is often necessary for a reader to contact multiple storage-nodes to ensure that it sees the most recent instance. Examples of decentralized storage systems that use this design include Harp [20], Petal [19], BFS [6], and Farsite [5].

Some decentralized storage systems spread data among storage-nodes more space-efficiently, using erasure coding or even simple striping. With striping, a data-item is divided into fragments, all of which are needed to reconstruct the entire data-item. With erasure coding, a data-item is encoded into a set of pieces such that any sufficient subset allows reconstruction. (A simple erasure code is the RAID mechanism [27] of striping plus parity computed across the stripe units.) With these data distribution schemes, reads require fragments from multiple servers. Moreover, the set of fragments must correspond to the same write operation. Examples of decentralized storage systems that use erasure coding include Zebra [12], SwiftRAID [21], Intermemory [8], and Myriad [7].

2.1 Related work

There has been much work on protocols for maintaining consistency across distributed servers. This section highlights some relevant related work.

A common approach to dealing with the partial writer problem is two-phase commit [11]. This works for both replication and erasure coding, but doubles the number of round-trips for every write. For replication, the partial writer problem can also be addressed via "repair," which involves a client or storage-node distributing a partially written value to storage-nodes that have not seen it. This is usually not an option for systems using erasure codes, since partial writes may not provide enough information to reconstruct the original data or the missing fragments.

A common approach to dealing with concurrency is to suppress it, either via locking or optimistic concurrency control [17]. Alternately, many systems (e.g., Harp [20] and Petal [19]) serialize their actions through a primary storage-node.

An alternate approach to handling both problems is to have the data stored on storage-nodes be immutable [29, 30]. By definition, this eliminates the difficulties of updates for existing data. In doing so, it shifts the problem up one level — an update now consists of creating a new data-item and modifying the relevant name to refer to it. Decoupling the data-item creation from its visibility simplifies both, but making the metadata service fault-tolerant often brings back the same issues. Examples of systems that use this model include SWALLOW [29], Amoeba [25], and most of the recent peer-to-peer filesystems (e.g., Past [31], CFS [9], Farsite [5], and the archival portion of Oceanstore [16]).

Our target consistency semantics (linearizability with read aborts) have been studied previously. Notably, Pierce [28] presents a protocol implementing these semantics in a decentralized storage system. This protocol is achieved by conjoining a protocol that implements pseudo-regular semantics (regular semantics [18] with read aborts) with a "write-back" protocol (repair). The penultimate step of a read operation is to write-back the intended return value, which ensures that the return value of the read operation is written to a full quorum before it is returned. The utility of write-back hinges on full data replication, allowing a reader to utilize the value fetched from a single storage-node (assuming no Byzantine failures). As noted earlier, this form of repair is not possible when data is distributed with more space-efficient schemes (e.g., erasure codes).

Versioning storage-nodes in our protocol provide capabilities similar to "listeners" in the recent work of Martin, et al. [22]. The listeners protocol guarantees linearizability (*without* read aborts) in a decentralized storage system. To do so, a read operation establishes a connection with a storage-node. The storage-node sends the current data-item value to the client. As well, the storage-node sends updates it receives back to the client, until the client terminates the connection. Thus, a reader may be sent multiple versions of a data item. In our protocol, readers look backward in time via the versioning storage-nodes, rather than listen into the future. Looking back in time is more message-efficient in the common case, and it avoids the need for repair (a.k.a., write-back) to deal with client failures. It does, however, come with the possibility of read aborts.

Many have exploited the fact that messaging overhead and round-trip counts can be reduced with loosely synchronized clocks. For example, Adya, et al. [1, 2] use loosely synchronized clocks as a component of a logical timestamp.

3 System Model

This section describes the system model assumed for the protocol presented, which provides linearizability with the possibility of read aborts in an asynchronous system. It introduces terminology, explains the capabilities of storage-nodes, defines the failure model, and details the consistency guarantee being pursued.

```
/* Write $V$ to data-item $D$ at logical time $LT$ */
$\langle S, LT \rangle :=$ WRITE $(D, V, LT)$

/* Read latest logical time $LT$ of data-item $D$ */
$\langle S, LT \rangle :=$ QUERY_TIME $(D)$

/* Read latest $V$ of data-item $D$ */
$\langle S, LT, V \rangle :=$ READ_LATEST $(D)$

/* Read $V$ of data-item $D$ previous to logical time $LT$ */
$\langle S, LT, V \rangle :=$ READ_PREVIOUS $(D, LT)$
```

D	= data-item ID
LT	= logical timestamp
S	= storage-node ID
V	= data-item value

Figure 2: **Interface of a versioning storage-node**

3.1 Terminology

Discussion of the system infrastructure is phrased in terms of *data-items*, *clients*, and *storage-nodes* (servers). There are N storage-nodes and any number of clients in the system. There are two types of *operations* in the protocol — *read operations* and *write operations* — both of which operate on *data-items*. Clients perform read/write operations that issue multiple read/write *requests* to storage-nodes. Requests are *executed* by storage-nodes; for example, a request that is executed by a storage-node is complete at that storage-node. A storage-node that executes a write request is said to *host* that write request or data-item. For clarity, and without loss of generality, we describe the protocol in terms of replication—thus, read/write requests transfer data-items from/to storage-nodes. By design, no aspect of the protocol assumes that storage-nodes hold an entire data-item rather than a data-item fragment (e.g., one portion of an erasure-encoded data-item).

Each data-item is uniquely identified by a *data-item ID* (e.g., a logical block number). A *client ID* uniquely identifies a client in the system. Each client maintains a *client request ID* that uniquely identifies each write operation by the client. Each write operation is distinguished by a unique *logical timestamp* of the form ⟨*data-item time, client ID, client request ID*⟩. Left to right precedence is used to compare two logical timestamps (i.e., data-item time is compared before client ID which is compared before client request ID). The client ID is included to distinguish between write operations from different clients at the same data-item time. The client request ID is included to distinguish between write operations from the same client at the same data-item time.

3.2 Versioning storage-nodes

Storage-nodes maintain versions of data-items. Every write request creates a new version of the data-item at the storage-node. Versions of a data-item are distinguished by logical timestamps (i.e., logical timestamps are unique per data-item).

Storage-nodes provide an interface to write a data-item at a specific logical time (`WRITE`). Indeed, the logical time of the write operation is a necessary parameter of a write request. Storage-nodes can execute *back-in-time* write requests, i.e., write requests with a logical timestamp earlier than the latest logical timestamp hosted by the storage-node. Back-in-time writes are inserted in timestamp order into the history of the data-item.

Storage-nodes provide an interface to read the latest data-item (`READ_LATEST`); an interface to read the latest data-item previous to a given timestamp (`READ_PREVIOUS`); and an interface to read the logical time of the latest data-item (QUERY_TIME). This interface is shown in Figure 2.

It is assumed that all versions of a data-item that are hosted at a storage-node are stored indefinitely. Realistically, it is necessary to periodically prune version histories to reclaim storage space. Determining which data-item versions can be safely pruned at each storage-node is non-trivial. The intricacies of version pruning are discussed in Section 6.3.

3.3 Failure Model

Communication between clients and storage-nodes is asynchronous, point-to-point, and reliable. Only client failures that occur during write operations need be considered, since those are the only ones that can affect other operations in the protocol. Clients may fail at any point during a write operation — before issuing any write requests, after issuing any number of write requests, or before receiving all write request acknowledgements. A *correct* client does not fail during a write operation.

Malicious (Byzantine) clients are not addressed here. However, it is possible for versioning storage-nodes to generate an audit trail and effectively roll-back malicious writes. Together, these features provide some protection from malicious clients [34]. Byzantine clients are briefly discussed in Section 6.2.

Storage-nodes may *crash*. For simplicity, we assume that crash failures are permanent. No more than t storage-nodes may fail.

3.4 Consistency Model

Operations are *concurrent* if their *operation durations* overlap. The duration of a read operation is measured from the start time at the issuing client to the end time at the client. A read operation that returns a value is said to *complete*. A read operation may abort without returning a value. The duration of a write operation is measured from its start time at the issuing client to its completion time, which is defined to be the moment that a write threshold of storage-nodes have executed the write request. Note that the completion time of a write operation is a system property — the client issuing a write operation will not know its exact completion time.

Operations are linearizable if there exists some sequential order of operations such that the order of operations at each client is maintained and each operation appears to occur at a distinct time within its operation duration [13].

The targeted consistency model is *linearizability with read aborts*, which is similar to Pierce's "pseudo-atomic consistency" [28]. That is, the set of all write operations and all complete read operations must be linearizable. Read operations that abort are excluded from the linearizable set of operations.

4 Decentralized Consistency via Versioning

This section presents the protocol for decentralized consistency via versioning storage-nodes. It also discusses a proof of correctness, the consequences of read and write threshold sizes, performance enhancements, and how it would be used in a synchronous environment.

4.1 Overview

At a high-level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations. At the start of a write operation, a client issues a QUERY_TIME operation to determine a data-item time that is greater than, or equal to, the data-item time of the *latest complete write* (the complete write with the highest timestamp). A logical timestamp is constructed that is guaranteed to be unique and greater than those seen from the previous QUERY_TIME operations. Logical timestamps identify write requests from the same write operation across storage-nodes. Logical timestamps also ensure that write-write concurrency cannot necessitate an abort, in contrast to optimistic concurrency control protocols that perform validation via two-phase commit.

For a read operation, a client issues read requests to the set of N storage-nodes. Once at least a read threshold of storage-nodes reply, the client identifies the *latest candidate write*, which is the data-item version returned with the greatest logical timestamp.

The latest candidate write is either part of the latest complete write or part of a *partial-write*. A partial-write is a write that has not yet completed at a write threshold of storage-nodes. If it is determined that the latest candidate write is a complete write, it is the latest complete write, and the read operation completes. If it is determined that the latest candidate write is a partial-write, it is discarded, previous data-item versions are requested, and a new latest candidate write is identified. This step may be repeated. The determination of complete and partial-writes is discussed in later sections. A read operation aborts if the client cannot determine if the latest candidate write is either complete or partial. Since it is possible for read operations to abort, the protocol can only guarantee linearizability with read aborts. The next three sections provide more detail on read and write thresholds, write operations, and read operations.

4.2 Read and Write Thresholds

There are constraints on the number of requests that must be executed for an operation to be complete. The write threshold W and the read threshold R are related to one another, as well as to N (the number of storage-nodes in the system) and t (the upper bound on the number of storage-node failures in the system).

The threshold values, like those in voting and quorum systems, are bound by an "overlap" constraint:

$$R + W > N$$

To classify the latest candidate write as the latest complete write, a read operation must observe the latest complete write in its entirety. This places a lower bound on R:

$$W \leq R$$

The failure model further constrains the threshold values:

$$W \leq R \leq N - t$$

$$t < W \leq N - t$$

The upper bounds $(N - t)$ on R and W ensure that even if there are t failures in the system, write operations will complete and that read operations *may* complete. The write threshold is constrained to be greater than t to ensure that a complete write persists in the face of t failures.

Before detailing the write operation, it is necessary to introduce one additional constraint, R_o (read overlap), such that:

$$R_o > N - W$$

With R_o replies to the QUERY_TIME operation, a client can construct a logical timestamp greater than that of the latest complete write. A complete write must be successfully executed at at least W storage-nodes, so that at least one of the R_o is guaranteed to overlap with W. Notice that R_o, unlike R is not constrained to be greater than or equal to W.

```
WRITE $(D, V)$ :
 1: /* Phase 1 */
 2: SEND (QUERY_TIME, $\langle D \rangle$) TO $\{S : S \in U\}$
 3: repeat
 4:     RECEIVE (QUERY_TIME_RESPONSE, $\langle S, LT \rangle$)
 5:     $Q := Q \cup \langle S, LT \rangle$
 6: until $(|Q| == R_o)$
 7: /* Phase 2 */
 8: $\widetilde{LT} := \langle \mathrm{INC}[\mathrm{MAX}[\{q.LT.T : q \in Q\}]], CID, RID \rangle$
 9: SEND (WRITE, $\langle D, V, \widetilde{LT} \rangle$) TO $\{S : S \in U\}$
10: repeat
11:     RECEIVE (WRITE_RESPONSE, $\langle S, \widetilde{LT} \rangle$)
12:     $A := A \cup \langle S, \widetilde{LT} \rangle$
13: until $(|A| == W)$
```

A	= set of write acknowledgements
CID	= client ID
D	= data-item ID
LT	= logical timestamp
$\widetilde{LT}$	= logical timestamp for write operation
Q	= set of time query responses
R_o	= read observation threshold
RID	= client request ID
S	= storage-node in U
T	= data-item time
U	= set of N storage-nodes in the system
V	= value of data-item
W	= write threshold

Figure 3: **Write Operation**

4.3 Write Operation

The write operation is divided into two phases. The first phase determines the current logical timestamp for the data-item. The second phase issues write requests to a set of storage-nodes until at least a write threshold of write requests have been acknowledged. Pseudo code for a write operation to data-item D is shown in Figure 3.

Phase 1: (Lines 2 - 6) The client issues QUERY_TIME requests to the N storage-nodes. Once the client receives replies from R_o storage-nodes, it is guaranteed to have observed a logical timestamp greater than, or equal to, that of the latest complete write of the data-item. Partial-writes and write operations that are concurrent to the query may contain timestamps later than that of the latest complete write.

Phase 2: (Lines 8 - 13) The client constructs a new data-item time by incrementing the maximum observed data-item time and then appending the client ID and a new request ID. The client issues write requests to each of the storage-nodes with the new logical timestamp. The client can consider the write complete once W storage-nodes have acknowledged the write requests.

4.4 Read Operation

The read operation also has two phases. The first phase consists of the client establishing an initial latest candidate write. The second phase consists of determining if the latest candidate write is a complete write. Pseudo code for a read operation to data-item D is shown in Figure 4.

Phase 1: (Lines 2 - 6) The client issues read requests for the most recent version of data-item D to all

```
$V$ := READ $(D)$ :
 1: /∗ Phase 1 ∗/
 2: SEND (READ_LATEST, $\langle D\rangle$) TO $\{S : S \in U\}$
 3: **repeat**
 4:     RECEIVE (READ_LATEST_RESPONSE, $\langle S, LT, V\rangle$)
 5:     $Q := Q \cup \langle S, LT, V\rangle$
 6: **until** $(|Q| == R)$
 7: /∗ Phase 2 ∗/
 8: **repeat**
 9:     $\widetilde{LT}$ :=MAX$[\{q.LT : q \in Q\}]$
10:     $\widetilde{V} := q.V : q \in Q \wedge q.LT == \widetilde{LT}$
11:     $\widetilde{W} := \{q.S : q \in Q \wedge q.LT == \widetilde{LT}\}$
12:     **if** $(|\widetilde{W}| \geq W)$ **then**
13:         RETURN $\widetilde{V}$
14:     **end if**
15:     **if** $(|\widetilde{W}| + (N - R) \geq W)$ **then**
16:         ABORT
17:     **end if**
18:     $Q := \{q : q \in Q \wedge q.S \notin \widetilde{W}\}$
19:     SEND (READ_PREVIOUS, $\langle D, \widetilde{LT}\rangle$) TO $\{S : S \in U - Q\}$
20:     **repeat**
21:         RECEIVE (READ_PREVIOUS_RESPONSE, $\langle S, LT, V\rangle$)
22:         $Q := Q \cup \langle S, LT, V\rangle$
23:     **until** $(|Q| == R)$
24: **until** (FALSE)
```

D	= data-item ID
LT	= logical timestamp
$\widetilde{LT}$	= logical timestamp of latest candidate write
N	= number of storage-nodes in the system
Q	= set of read responses being considered
S	= storage-node in server-group U
t	= upper bound on failures in the system
U	= set of N storage-nodes in the system
V	= value of data-item
$\widetilde{V}$	= value of latest candidate write
W	= write threshold
$\widetilde{W}$	= subset of Q that host the latest candidate write

Figure 4: **Read Operation**

N storage-nodes. The client adds each response to a set of considered read responses (Q). Once the client's set contains R responses, the client identifies the latest candidate write.

Phase 2: (Lines 8 - 24) With R responses, the read operation can attempt to determine if the latest candidate write is a complete write or a partial-write. However, there is a chance that the latest candidate write cannot be classified as either complete or partial. This may occur if some of the storage-nodes that have not responded contain copies of the latest candidate write.

If the latest candidate write is a complete write, it is the latest complete write; the read operation completes by returning this value. If the latest candidate write cannot be classified, the read operation aborts. If the latest candidate write is an identifiable partial-write, it is discarded.

All storage-nodes that host the discarded latest candidate write are removed from the set of read responses. Requests for the data-item version previous to the discarded latest candidate write are issued to all storage-nodes not currently in the set of read responses. Once the set of read responses contains R responses,

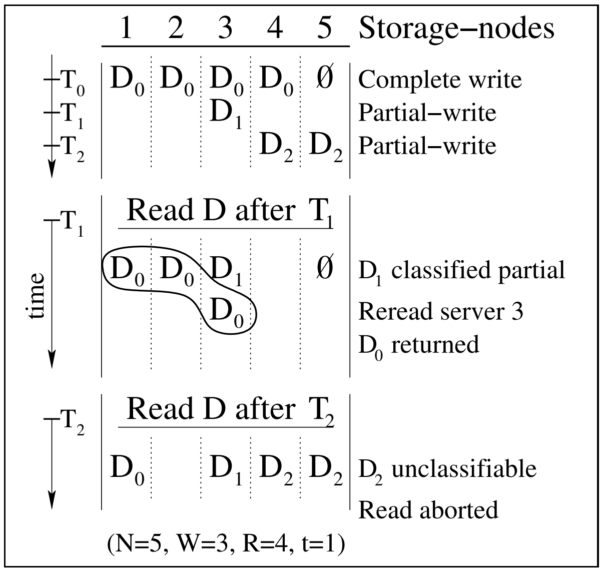

Figure 5: **Asynchronous Read Example:** This figure shows the steps necessary to perform a read operation of data-item D (D_i represents the version of D at logical time i) for each of two reads. The read after time T_1 successfully completes in two steps. The read after time T_2 spuriously aborts.

the second phase begins anew. The read operation continues in this manner until the latest complete write is identified or the read operation aborts.

4.4.1 Example read operation

An example illustrating the steps a read operation performs is shown in Figure 5. The first read, at time T_1, successfully completes. The initial latest candidate write, D_1, is classified as a partial-write and discarded. The client can deduce this, since even if storage-node four contained D_1, a write threshold for D_1 does not exist. Read requests for the next latest candidate write (D_0) are issued, and the responses are sufficient to identify it as the latest complete write.

The second read (at time T_2) must abort due to a lack of information. The latest candidate write, D_2, cannot be classified as partial or complete. This read is different from the first, in that storage-node two may contain D_2. If it does, D_2 would be the latest complete write. The asynchronous communication model precludes issuing a read request to storage-node two, since it may be failed. The read operation must abort.

4.5 Proof of Correctness

In this section we sketch a proof that our protocol implements linearizability of operations [13], excluding reads that abort.

Definition 1. *Operation o_1 precedes operation o_2 if operation o_1 completes before operation o_2 begins. The precedence relation is written as $o_1 \to o_2$.*

Operation o_2 is said to follow, or to be subsequent to, operation o_1. The notation $o_1 \not\to o_2$ is used to mean operation o_1 does not precede operation o_2; o_1 is either concurrent to o_2 or $o_2 \to o_1$. Also, note that a partial-write does not precede any operation, since it never completes.

Definition 2. *$T(w_i)$ denotes the logical timestamp of write operation i.*

Definition 3. *The notation $r_i(w_j)$ means that read operation i returns the value (first component) of the ⟨data-item value, logical timestamp⟩ pair written by write operation j.*

For a read operation to return a value, the read operation must complete. Aborted read operations are excluded from the set of read operations for which r_i is defined.

Lemma 1. *If $w_x \to w_y$, then $T(w_x) < T(w_y)$.*

Proof sketch: $R_o > N - W$ ensures that the first phase of a write operation achieves a higher timestamp than any other write operation that preceded it. □

Observation 1. $T(w_x) == T(w_y) \iff x == y$. Uniqueness of logical timestamps is guaranteed by the inclusion of the client ID and the client request ID.

Observation 2. *Timestamp order is a total order on write operations that is consistent with the precedence order among writes.*

Lemma 1 proves that timestamp order is consistent with precedence. And, by Observation 1, timestamp order does impose a total order on all writes.

Lemma 2. *If $r_p(w_x) \wedge r_q(w_y) \wedge r_p \to r_q$, then $T(w_x) \le T(w_y)$.*

Proof sketch: Since $r_p(w_x)$, read r_p observed at least W storage-nodes holding the value and timestamp of w_x. Since during r_q, at most t of these servers could be unavailable to r_q, r_q must interact with at least $W - t \ge W - (N - R)$ servers that hold the value and timestamp of w_x. Since r_q does not abort, it must observe W servers holding the value and timestamp of w_x or holding a write with a higher timestamp. □

Note that there is a partial order on all read operations (that complete), obtained by ordering reads according to the timestamps of the write operations whose values they return, i.e., $r_p(w_x) \prec r_q(w_y) \iff T(w_x) < T(w_y)$. Lemma 2 ensures that this partial order is consistent with precedence among reads, and so any way of extending this partial order to a total order yields an ordering of reads that complete that is consistent with precedence among reads.

Lemma 3. *If $r_p(w_x) \wedge T(w_x) < T(w_y)$, then $w_y \not\to r_p$.*

Proof sketch: Assume for a contradiction that $w_y \to r_p$. Since w_y is complete when r_p begins, r_p must return the result of a write operation with a logical timestamp greater than, or equal to, w_y (as in Lemma 2). However, r_p returns w_x, a contradiction. Thus, $w_y \not\to r_p$. □

Lemma 3 provides the basis for "merging" the timestamp order on writes with the timestamp order of (the values returned by) complete reads to obtain a totally ordered set of operations consistent with precedence, since it shows that each read can be ordered following the write whose value it returns (and before any later write in timestamp order). This completes the argument of linearizability for all operations except aborted reads.

4.6 Refinements For Implementations

Finding the maximal current timestamp: It is possible for a write operation to begin and complete such that it is ordered behind a partial-write that existed before the operation began. The base protocol waits for only R_o responses to its `QUERY_TIME` requests, which is sufficient for correctness. But, waiting for more storage-nodes to reply would increase the probability that a logical timestamp is constructed that is greater than that of an earlier partial-write.

Observability of complete writes: Whenever the latest candidate write is unclassifiable, a read must abort. This can occur for both partial-writes and complete writes. A partial-write that exists at $[W - t, W - 1]$ storage-nodes may not be classifiable as partial. A complete write that exists at $[W, W + t - 1]$ storage-nodes may not be classifiable as complete. Since t storage-nodes may be failed, fewer than W of the storage-nodes hosting the complete write could be available. In this scenario, it is possible for the write operation to be unclassifiable and for the above cases to be indistinguishable to a read operation. In order to maximize the likelihood of subsequent reads observing a complete write, it can be useful to write to more storage-nodes than necessary.

Spurious aborts and retries: The asynchronous system model may lead to spurious read aborts. The problem lies in the fact that only $R \leq N - t$ read responses are collected before attempting to distinguish a partial-write from a complete write. In the case of $R < N - t$, it might be useful to wait until $N - t$ responses are collected, since this reduces the probability of aborting. Moreover, it is possible — in many systems likely — that there are fewer than t failures in the system. If responses from greater than $N - t$ storage-nodes could be solicited, read operations would complete more often. However, the asynchronous system model precludes such an approach. In some cases, retrying an aborted read operation will result in successful completion. For example, responses from a different set of $N - t$ storage-nodes (e.g., due to variations in response times) could allow the read operation to complete.

Early completion of reads: The base protocol waits for R responses to its `READ_LATEST` requests. But, a read operation may be able to complete correctly before R read responses are received, if $R >$MAX$[W, N - W + 1]$. That is, read operations can be made more efficient if the set of responses is examined as each arrives, once overlap is guaranteed, in order to determine if the latest candidate write can be classified.

4.7 Synchronous system model

Placing a bound on the message propagation delay changes the properties of the consistency protocol. Refinements that take advantage of a synchronous system model can be introduced. Since synchrony assumptions (i.e., timeouts) are often employed in real implementations, it is useful to consider the consistency protocol in a synchronous system model.

A synchronous system model reduces the amount of concurrency in a system by bounding the duration of correct write operations. More importantly, a synchronous model allows read operations to consider more than $N - t$ responses before having to abort. The exit condition from the repeat-until loop is modified to collect read responses until either a timeout occurs or all storage-nodes respond. This provides a read operation with responses from all correct storage-nodes. Such information can reduce the frequency of spurious aborts. In an asynchronous system model, the client cannot use timeouts to deduce the set of failed storage-nodes.

5 Loosely Synchronized Clocks

This section examines an alternate form of the "consistency via versioning" protocol, which uses loosely synchronized client clocks to determine the data-item time component of the logical timestamps. Protocols to achieve approximate clock synchronization in today's networks are well known, inexpensive, and widely deployed [23, 24]. In the resulting consistency protocol, a write operation requires significantly fewer messages than in the original protocol. Clock synchronization messages are required, but they are out-of-band from read and write operations. The consistency model remains the same (linearizability with read aborts). But, the definition of operation duration has to be expanded to accommodate clock skew.

5.1 Protocol Modifications

The original protocol's write operation is modified to use the client's local clock; that is, Phase 1 of Figure 3 is replaced by a simple read of the client's local clock. The local clock is used as the data-item time in the logical timestamp for a write operation. With this change a `QUERY_TIME` operation becomes unnecessary, and there are one fewer roundtrips per write operation. The logical timestamp in the refined protocol is still guaranteed to be unique for each data-item, because of the client-specific fields, and it serves the same function.

Client clocks are not perfectly synchronous — there is some skew, bound by θ, between a client clock and global real time. Thus, data-item times based on client clocks result in a new definition of operation

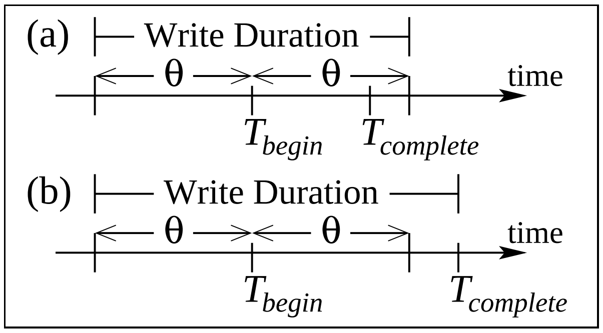

Figure 6: **Write Duration** This figure shows the durations of two different write operations. For each, the operation duration extends backwards by θ. Each write begins at time T_{begin} and complete at time $T_{complete}$. Figure (a) shows the operation duration extended forward to θ. Figure (b) shows the operation duration ending at the operation completion time.

duration for write operations. The clock skew extends each write operation duration by up to 2θ. More precisely, to account for clock skew, the duration of the write operation is padded by the upper bound on clock skew. Thus, if the write operation begins at global time T_{begin}, its duration is at least from $T_{begin} - \theta$ to $T_{begin} + \theta$. Assuming that the write operation completes at $T_{complete}$, its duration is from $T_{begin} - \theta$ (θ time units behind the local clock) to the maximum of $[T_{complete}, T_{begin} + \theta]$. An example of this is shown in Figure 6. The definition of "concurrent operation" and the consistency guarantee provided by the protocol does not change. But, longer operation durations can lead to more operations being defined as concurrent to one another.

A synchrony assumption for the exchange of clock messages is required to bound θ. No other synchrony assumption need be introduced to the system model. For example, the synchronous clock sub-system may use a synchronous communication path (e.g., GPS), while the consistency protocol operates over an asynchronous communication path.

5.2 Consequences of Clock Skew

Pseudo-write sharing: Clock skew introduces *pseudo-write sharing* for write operations by distinct clients that begin within 2θ of each other. This pseudo-write sharing can cause the reader-observed ordering of write operations to differ from the real-world order. In particular, a write operation by a client with a slow clock may remain after a "subsequent" write operation by a client with a fast clock — the client with the fast clock may write backwards in logical time. Writes that begin more than 2θ apart in global time are guaranteed to be correctly ordered by the precedence relation. Clearly, a system with small θ provides a more usable consistency model than one with a large θ.

Lost read-modify-write sequences: Pseudo-write sharing introduces a problem for read-modify-write sequences of operations. The problem is best illustrated through an example. A client performs a read of data-item D. The latest complete write of D, w_1, has data-item time T_1. The client modifies the value read and writes it back as w_2 with data-item time T_2 (the local clock value). If $T_2 < T_1$, the second write is written back in time, even though the initial write clearly completed first.

This problem can be solved when a single client performs both the read and the write operation. To do so, a client must keep track of the data-item times of read operations and utilize them in write operations during a read-modify-write sequence. Specifically, if the local client clock is behind the latest data-item time returned in the read operation, the client instead increments and uses the data-item time from the read operation. In this manner, the client guarantees that the write operation occurs later in logical time than the write operation observed by the read operation. Note that as long as the previous write's data-item time is within θ of global-time and the read-modify-write cycle takes a non-zero amount of time, the incremented

Protocol	**Operation**	**Transmit Msgs**	**Reply Msgs**	**Total Msgs**
Asynchronous	Write	$[R,N]+[W,N]$	$[R,N]+[W,N]$	$[2R+2W,4N]$
	Read	$[R,N]$	$[R,N-t]$	$[2R,2N-t]$
Synchronous	Write	$[R,N]+[W,N]$	$[R,N]+[W,N]$	$[2R+2W,4N]$
	Read	$[R,N]$	$[R,N]$	$[2R,2N]$
Synchronized Clocks	Write	0 + $[W,N]$	0 + $[W,N]$	$[2W,2N]$
2-Phase-Commit	Write	$N+N$	N	$3N$
	Read	$[1,N]$	$[1,N]$	$[2,2N]$
Listeners [22]	Write	$[W,N]+[W,N]$	$[W,N]+[W,N]$	$[4W,4N]$
	Read	$R+N$	$[W,R]$	$[R+W+N,2R+N]$

Table 1: **Number of messages exchanged in non-conflict, no-failure case:** The notation $[X,Y]$ expresses a bound on the number of messages such that the lower bound X guarantees correctness and the upper bound Y may be useful in certain cases (e.g., improved read availability). If two sets of bounds are provided (e.g., in the write cases), each set of bounds refers to a different phase of that operation.

data-item time is also within θ of global-time.

Unfortunately, this solution does not extend to read-modify-write sequences involving out-of-band communication. For example, client A reads data-item D and sends the value to client B, who then performs a write operation based on the value of D. A possible solution would be to extend storage-nodes to return the logical timestamp of the latest write request to a data-item with write request responses. This hint could be used by a client to determine if its write is the latest write seen by a particular storage-node. For example, by comparing the set of timestamps returned by all storage-nodes, the client can determine whether the write operation would be viewed as the latest complete write. As a form of optimistic concurrency control [17], it could then retry the sequence.

6 Discussion

6.1 Performance

Table 1 shows the bounds on the number of messages that the consistency protocols discussed in this paper, together with two-phase commit and the Listener's protocol described in [22], must exchange in common cases of no failures and no concurrency. Bounds on the number of messages are given, such that the lower bound guarantees correctness, while the upper bound gives the worst-case (which may be useful in certain situations, e.g., improved read availability).

In common mode operation, there are no failures and a low degree of write-write and read-write concurrency. Thus, very few partial-writes are observed. Using the asynchronous model, clients in this mode are expected to execute write requests at $N-t$ storage-nodes and read requests at R storage-nodes. Using the synchronous model, clients in this mode are expected to execute write requests at all non-failed (between N and $N-t$) storage-nodes and read requests at R storage-nodes. Recall that writes go to more than W nodes to minimize aborts and maximize availability. If clients are using schemes such as erasure codes or striping, the protocol adds no excess communication, since it is usually appropriate for clients to communicate with approximately the same number of storage-nodes.

Protocols that ensure no partial-writes ever occur, such as two-phase commit, have higher per-write operation costs and may have lower per-read operation costs for replication. For erasure codes or striping, however, this would not be the case. The Listeners protocol has extra round-trip messages that are required to setup and tear-down state that the clients maintain on the storage nodes.

6.2 Byzantine Clients

Malicious clients are difficult. Of course, a malicious client can corrupt any data to which it has write access. As well, a malicious client can deliberately perform a partial-write, or even write mismatching data-items to different storage-nodes. The protocol correctly handles the former, however performance may degrade. Solving the latter problem will require the addition of data integrity checks to the protocol.

Using loosely synchronized client clocks to improve write efficiency introduces another attack. A malicious client can write data-items with future timestamps. If this occurs, subsequent updates by correct clients may be lost since they become “back-in-time” updates. If storage-nodes are assumed to have clocks that are loosely synchronized with the client clocks, then storage-nodes can disallow write requests with data-item times more than 2θ ahead of the storage-node clock. Such a policy bounds the distance into the future a malicious client may write.

Finally, the versioning performed by the storage-nodes can be leveraged to diagnose and recover from malicious client actions [34]. For example, since the storage-nodes keep a full history of every data-item, it is possible to retrieve old versions (that occurred before the malicious writes) and copy them forward to the present. Automating such recovery for a decentralized store is an interesting avenue for future research.

6.3 Old Version Pruning

Eventually, old versions must be pruned to prevent the exhaustion of storage-node capacity. But only data-items previous to the latest complete write may be safely pruned, and the completeness of a write operation is not observable by a single storage-node.

Pruning is envisioned to be controlled by a garbage collection service run above the storage service. The garbage collector can identify the latest complete write by performing a read operation that completes. To prune data-items previous to the latest complete write, a garbage collector interface is added to the storage-node to allow deletion of all versions of data-items with logical timestamps before a given one (e.g., the latest complete write).

7 Summary

Versioning storage-nodes enable a consistency protocol that is efficient in the common case of no update conflicts and no failures. Moreover, versioning storage-nodes allow the protocol to support erasure-coded data with client failures (i.e., partial-writes of erasure-coded data are non-destructive). The protocol described in this paper guarantees linearizability with the possibility of read aborts. The protocol is correct in an asynchronous environment and is readily optimized in a synchronous environment.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1995.

[2] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. *ACM Symposium on Principles of Distributed Computing* (Santa Barbara, CA, August 1997), pages 73–82. ACM, 1997.

[3] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 298–307. IEEE Computer Society, 2000.

[4] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991.

[5] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *ACM SIGMETRICS Performance Evaluation Review*, **28**(1):34–43. ACM Press, 2000.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 173–186. ACM, 1998.

[7] Fay Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon Perl, and Li Zhang. Myriad: cost-effective disaster tolerance. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 103–116. USENIX Association, 2002.

[8] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival Intermemory. *ACM Conference on Digital Libraries* (Berkeley, CA, 11–14 August 1999), pages 28–37. ACM, 1999.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, AB, Canada, 21–24 October 2001). Published as *Operating System Review*, **35**(5):202–215, 2001.

[10] Deepinder S. Gill, Songian Zhou, and Harjinder S. Sandhu. *A case study of file system workload in a large–scale distributed environment*. Technical report CSRI–296. University of Toronto, Ontario, Canada, March 1994.

[11] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[12] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, **13**(3):274–310. ACM Press, August 1995.

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

[14] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[15] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, **10**(1):3–25. ACM Press, February 1992.

[16] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaten, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):190–201, 2000.

[17] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, **6**(2):213–226, June 1981.

[18] Leslie Lamport. *On interprocess communication*. Technical report 8. Digital Equipment Corporation, Systems Research Center, Palo Alto, Ca, December 1985.

[19] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[20] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. *ACM Symposium on Operating System Principles* (Pacific Grove, CA, 13–16 October 1991). Published as *Operating Systems Review*, **25**(5):226–238, 1991.

[21] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. Swift/RAID: a distributed RAID system. *Computing Systems*, **7**(3):333–359. Usenix, Summer 1994.

[22] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. *International Symposium on Distributed Computing* (Toulouse, France, 28–30 October 2002), 2002.

[23] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE-ACM Transactions on Networking*, **3**(3), June 1995.

[24] David L. Mills. *Network time protocol (version 3)*, RFC–1305. IETF, March 1992.

[25] Sape J. Mullender. A distributed file service based on optimistic concurrency control. *ACM Symposium on Operating System Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):51–62, December 1985.

[26] Brian D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994). Published as *Performance Evaluation Review*, **22**(1):138–149. ACM, 1994.

[27] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[28] Evelyn Tumlin Pierce. *Self-adjusting quorum systems for byzantine fault tolerance*. PhD thesis, published as Technical report CS–TR–01–07. Department of Computer Sciences, University of Texas at Austin, March 2001.

[29] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. *International Workshop on Local Networks* (Zurich, Switzerland), August 1980.

[30] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, **1**(1):3–23. ACM Press, February 1983.

[31] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, AB, Canada, 21–24 October 2001). Published as *Operating System Review*, **35**(5):188–201. ACM, 2001.

[32] Douglas J. Santry, Michael J. Feeley, and Norman C. Hutchinson. Elephant: the file system that never forgets. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1992). IEEE Computer Society, 1999.

[33] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. *Metadata efficiency in a comprehensive versioning file system*. Technical report CMU–CS–02–145. Carnegie Mellon University, 2002.

[34] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[35] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.