The safety and liveness properties of a protocol family for versatile survivable storage infrastructures

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

CMU-PDL-03-105
March 2004

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Survivable storage systems mask faults. A protocol family shifts the decision of which types of faults from implementation time to data-item creation time. If desired, each data-item can be protected from different types and numbers of faults with changes only to client-side logic. This paper presents proofs of the safety and liveness properties for a family of storage access protocols that exploit data versioning to efficiently provide consistency for erasure-coded data. Members of the protocol family may assume either a synchronous or asynchronous model, can tolerate hybrid crash-recovery and Byzantine failures of storage-nodes, may tolerate either crash or Byzantine clients, and may or may not allow clients to perform repair. Additional protocol family members for synchronous systems under omission and fail-stop failure models of storage-nodes are developed.

Acknowledgements: We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, 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. This work is also supported by the Army Research Office through grant number DAAD19-02-1-0389 to CyLab at Carnegie Mellon University. Garth Goodson was supported by an IBM Fellowship.

Keywords: survivable storage, Byzantine fault-tolerance, crash-recovery failures, omission failures, fail-stop failures, hybrid failure models, atomic registers, erasure codes

1 Introduction

Survivable, or fault-tolerant, storage systems protect data by spreading it redundantly across a set of storage-nodes. In the design of such systems, determining which kinds of faults to tolerate and which timing model to assume, are important and difficult decisions. Fault models range from crash faults to Byzantine faults [16] and timing models range from synchronous to asynchronous. These decisions affect the access protocol employed, which can have a major impact on performance. For example, a system's access protocol can be designed to provide consistency under the weakest assumptions (i.e., Byzantine failures in an asynchronous system), but this induces potentially unnecessary performance costs. Alternatively, designers can "assume away" certain faults to gain performance. Traditionally, the fault model decision is hard-coded during the design of the access protocol.

This traditional approach has two significant shortcomings. First, it limits the utility of the resulting system—either the system incurs unnecessary costs in some environments or it cannot be deployed in harsher environments. The natural consequence is distinct system implementations for each distinct fault model. Second, all data stored in any given system implementation must use the same fault model, either paying unnecessary costs for less critical data or under-protecting more critical data. For example, temporary and easily-recreated data incur the same overheads as the most critical data.

In [9], we promote an alternative approach, in which the decision of which faults to tolerate is shifted from design time to data-item creation time. This shift is achieved through the use of a *family* of access protocols that share a common server implementation and client-server interface. A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by simply changing the number of storage-nodes utilized, and some read and write thresholds. A protocol family enables a given infrastructure of storage-nodes to be used for a mix of fault models and number of faults tolerated, chosen independently for each data-item.

The protocol family covers a broad range of fault model assumptions (crash-recovery vs. Byzantine servers, crash vs. Byzantine clients, synchronous vs. asynchronous communication, client repairs of writes vs. not, total number of failures) with no changes to the client-server interface or server implementation. Protocol family members are distinguished by choices enacted in client-side software: the number of storage-nodes that are written and the logic employed during a read operation.

In this paper, we identify and prove the safety and liveness properties that each member of the protocol family achieves.

The remainder of this paper is organized as follows. Section 2 describes our protocol family. Section 3 describes the mechanisms employed by the protocol family to protect against Byzantine faults. Section 4 details how asynchronous protocol members are realized within a common software implementation. Section 5 develops constraints (e.g., on the minimum number of storage-nodes required) for asynchronous members. Sections 6 and 7 respectively prove the safety and liveness properties of asynchronous protocol family members. As well, the distinct safety and liveness properties of each protocol family member are identified. Section 8 extends the development of asynchronous protocol members into the synchronous timing model, yielding the synchronous protocol members. Moreover, additional protocol family members for synchronous environments that tolerate omission failures and fail-stop failures instead of crash-recovery failures are developed. Each distinct storage-node failure model for synchronous protocol family members leads to distinct constraints (e.g., on the minimum number of storage-nodes required). In Section 9 synchronous protocol members are extended to take advantage of loosely synchronized client clocks. Section 10 discusses work related to members of the protocol family.

2 The protocol family

We describe each protocol family member in terms of N storage-nodes and an arbitrary number of clients. There are two types of operations — reads and writes. Each read/write operation involves read/write requests from a client to some number of storage-nodes. We assume that communication between a client and a storage-node is point-to-point and authenticated; channel reliability is discussed in Section 2.1.2.

At a high level, the protocol proceeds as follows. A data-item is *encoded* into data-fragments; any threshold-based erasure code (e.g., replication, secret sharing [26], information dispersal [23], short secret sharing [15]) could be used. Logical timestamps are used to totally order all write operations and to identify data-fragments from the same write operation across storage-nodes. For each correct write, a client constructs a logical timestamp that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). A write operation is *complete* once sufficient benign (non-Byzantine) storage-nodes have executed write requests. The exact number of storage-nodes that must execute a write request for a client to know that its write operation is complete differs among protocol members. Storage-nodes provide fine-grained versioning; a correct storage-node stores a data-fragment version (indexed by logical timestamp) for each write request it executes.

To perform a read operation, a client issues read requests to a set of storage-nodes. From the responses, the client identifies the *candidate*, which is the data-fragment version returned with the greatest logical timestamp. The read operation *classifies* the candidate as complete, incomplete or unclassifiable based on the number of read responses that share the candidate's timestamp. If the candidate is classified as *complete*, then the read operation is complete; the value of the candidate is returned. If it is classified as *incomplete*, the candidate is discarded, another read phase is performed to collect previous versions of data-fragments, and classification begins anew; this sequence may be repeated. If the candidate is *unclassifiable*, members of the protocol do one of two things: repair the candidate or abort the read operation.

2.1 Family members

Each member of the protocol family is characterized by four parameters: the timing model, the storage-node failure model, the client failure model, and whether client repair is allowed. Eight protocol members result from the combination of these characteristics, each of which supports a hybrid failure model (crash-recovery and Byzantine) of storage-nodes.

2.1.1 Timing model

Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions (i.e., no assumptions about message transmission delays or execution rates). In contrast, synchronous members assume known bounds on message transmission delays between correct clients/storage-nodes and their execution rates.

2.1.2 Storage-node failure model

Family members are developed with a hybrid storage-node failure model [27]. Under a traditional hybrid failure model, up to t storage-nodes could fail, $b \leq t$ of which may be Byzantine faults; the remainder could only crash. However, we consider a hybrid failure model for storage-nodes that crash and recover.

First, we review the crash-recovery model from Aguilera et al. [1]. In a system of n processes, each process can be classified as always-up, eventually-up, eventually-down, or unstable. A process that is *always-up* never crashes. A process that is *eventually-up* crashes at least once, but there is a time after which it is permanently up. A process that is *eventually-down* crashes at least once, and there is a time after which it is permanently down. A process that is *unstable* crashes and recovers infinitely many times. These classifications are further refined: a process is *good* if it is either always-up or eventually-up.

We combine the crash-recovery model with the hybrid failure model as follows. Up to b storage-nodes may ever be Byzantine; such storage-nodes do not recover and are not *good*. There are at least $N-t$ good storage-nodes (where $b \leq t$). A storage-node that is not Byzantine is said to be *benign* (i.e., benign storage-nodes are either always-up, eventually-up, eventually-down, or unstable).

We assume that storage-nodes have stable storage that persists throughout the crash and recover process. We assume point-to-point authenticated channels with properties similar to those used by Aguilera et al. [1]. In summary, channels do not create messages (*no creation*), channels may experience *finite duplication*, and channels are *fair loss*. The finite duplication property ensures that if benign process p sends a message to benign process q only a finite number of times, then q receives the message only a finite number of times. The fair loss property ensures that if benign process p sends infinitely many messages to good process q, then q receives infinitely many messages from p.

The timing model and storage-node failure model are interdependent. In an asynchronous system, storage-node crashes are indistinguishable from slow communication. In a synchronous system, storage-nodes that crash could be detectable via timeouts (i.e., the storage-nodes could fail-stop). However, in a crash-recovery failure model, the "fact" that a storage-node has timed out cannot be utilized; the timeout could be from a storage-node that, in the future, may respond. The crash-recovery failure model is a strict generalization of the omission and crash failure models. Under less general failure models, the lower bound on the total number of storage-nodes required is reduced for synchronous protocol members. As such, we consider synchronous protocol members under omission and fail-stop storage-node failure models in Section 8 .

2.1.3 Client failure model

Each member of the protocol family tolerates crash client failures and may additionally tolerate Byzantine client failures. We refer to non-Byzantine clients as benign (i.e., benign clients are either correct or crash). Crash failures during write operations can result in subsequent read operations observing an incomplete or unclassifiable write operation.

As in any general storage system, an authorized Byzantine client can write arbitrary values to storage. Byzantine failures during write operations can additionally result in a write operation that lacks integrity; the decoding of different sets of data-fragments could lead to clients observing different data-items. Mechanisms for detecting any such write operation performed by a Byzantine client are described in Section 3. These mechanisms successfully reduce Byzantine actions to either being detectable or crash-like, allowing Byzantine clients to be tolerated without any change to the thresholds.

The timing model and client failure model are interdependent. In an asynchronous system, readers cannot distinguish read-write concurrency from a crash failure during a write operation. In a synchronous system, readers can distinguish read-write concurrency from a crash failure during a write operation (by issuing multiple read requests separated in time by the known bound on write operation duration). However, in this paper, we do not consider synchronous protocol members that take advantage of this information.

2.1.4 Client repair

Each member of the protocol family either allows, or does not allow, clients to perform repair. Repair enables a client that observes an unclassifiable (i.e., *repairable*) candidate during a read operation to perform a write operation, which ensures that the candidate is complete, before it is returned (see Section 4.2.2).

In systems that differentiate write privileges from read privileges, client repair may not be possible. Non-repair protocol members allow read operations to abort. Reads can be retried at either the protocol or

Timing model	Client failure	Repairability	Safety	Liveness
Asynchronous	Crash-only	Repairable	Linearizable [12]	Wait-free [10, 14]
		Non-repair	Linearizable with read aborts [22]	Single-client wait-free
	Byzantine	Repairable	Byzantine-operation linearizable	Wait-free
		Non-repair	Byzantine-operation linearizable with read aborts	Single-client wait-free
Synchronous	Crash-only	Repairable	Linearizable	Wait-free
		Non-repair	Linearizable with read aborts	Single-client wait-free
	Byzantine	Repairable	Byzantine-operation linearizable	Wait-free
		Non-repair	Byzantine-operation linearizable with read aborts	Single-client wait-free

Table 1: **Safety and liveness properties of protocol family members.** For details on safety guarantees see Section 6.1. For details on liveness guarantees see Section 7.1.

application level. At the protocol level, concurrency is often visible in the timestamp histories—an aborted read could be retried until a stable set of timestamps is observed. Other possibilities include requiring action by some external agent or blocking until a new value is written to the data-item (as in the "Listeners" protocol of Martin et al. [19]).

2.2 Protocol guarantees

Each member of the protocol family has distinct safety and liveness properties. In Table 1, the guarantees made by protocol family members are summarized. Safety guarantees are discussed in Section 6.1 and Liveness guarantees are discussed in Section 7.1. The safety and liveness properties achieved are for the hybrid crash-recovery failure model of storage-nodes. Since the hybrid crash-recovery failure model is a strict generalization of the hybrid omission failure model and hybrid crash failure model, these safety and liveness properties hold in less general storage-node failure models. The liveness guarantees assume no storage exhaustion on storage-nodes.

3 Mechanisms

This section describes mechanisms employed for encoding data, and preventing Byzantine clients and storage-nodes from violating consistency. We assume that storage-nodes and clients are computationally bounded such that cryptographic primitives can be effective. Specifically, we make use of collision-resistant hash functions. As well we assume that communication is authenticated.

3.1 Erasure codes

We consider only threshold erasure codes in which any m of the N encoded data-fragments can decode the data-item. Moreover, every m data-fragments can be used to deterministically generate the other $N - m$ data-fragments. Example threshold erasure codes are replication, Shamir's secret sharing [26], Rabin's information dispersal [23], and Krawczyk's short secret sharing [15].

3.2 Data-fragment integrity

Byzantine storage-nodes can corrupt their data-fragments, which we informally refer to as an "integrity fault". As such, it must be possible to detect and mask up to b storage-node integrity faults. Cross checksums [6] are used to detect corrupt data-fragments: a cryptographic hash of each data-fragment is computed, and the set of N hashes are concatenated to form the *cross checksum* of the data-item. The cross checksum is stored with each data-fragment, enabling corrupted data-fragments to be detected by clients performing read operations (see Section 4.2.2).

3.3 Write operation integrity

Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates random data-fragments (rather than erasure coding a data-item correctly), then each of the $\binom{N}{m}$ subsets of data-fragments could "recover" a distinct data-item. This attack is similar to *poisonous writes* for replication, as described by Martin et al. [19]. To protect against such Byzantine client actions, read operations must only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in three ways:

Validating timestamps. To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.

Storage-node verification. On a write, each storage-node verifies its data-fragment against the corresponding hash in the cross checksum. The storage-node also verifies that the cross checksum matches the low-order bits of the validating timestamp. A correct storage-node only executes write requests for which both checks pass. Thus, a Byzantine client cannot make a correct storage-node appear Byzantine—only Byzantine storage-nodes can return unverifiable responses.

Validated cross checksums. Combined, storage-node verification and validating timestamps ensure that the data-fragments being considered by any read operation were not fabricated by Byzantine storage-nodes. To ensure that the client that performed the write operation acted correctly, the cross checksum must be validated by the reader. For the reader to validate the cross checksum, all N data-fragments are required. Given any m data-fragments, the reader can generate the full set of N data-fragments a correct client should have written. The reader can then compute the "correct" cross checksum from the generated data-fragments. If the generated cross checksum does not match the validated cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that can be validated.

4 The protocol family design

This section presents the protocol family design in the form of pseudo-code with supporting text for explanation. The pseudo-code relies on some new terms. The symbol, ts, denotes logical timestamp and, $ts_{\text{candidate}}$, denotes the logical timestamp of the candidate. The set, $\{D_1, \ldots, D_N\}$, denotes the N data-fragments; likewise, $\{S_1, \ldots, S_N\}$ denotes the N storage-nodes. A cross checksum is denoted CC. The operator '|' denotes concatenation.

The symbols, `COMPLETE` and `INCOMPLETE`, used in the read operation pseudo-code, are defined in Section 5. The rules for classifying an operation as complete or incomplete differ among protocol family members.

```
INITIALIZE() :
/* Each member of the history is a ⟨ logical time, data value, cross checksum ⟩ triple */
/* History is stored in stable storage */
History := ⟨0, ⊥, ⊥⟩

RECEIVE_TIME_REQUEST() :
SEND(TIME_RESPONSE, S, MAX[History.ts])

RECEIVE_WRITE_REQUEST(ts, D, CC) :
if (VALIDATE(ts, D, CC)) then
   /* Execute the write request */
   History := History ∪ ⟨ts, D, CC⟩
   SEND(WRITE_RESPONSE, S)
end if

VALIDATE(ts, D, CC) :
if ((HASH(CC) ≠ ts.Verifier) OR (HASH(D) ≠ CC[S])) then
   return (FALSE)
end if
/* Accept the write request */
return (TRUE)

RECEIVE_READ_LATEST() :
/* Note, Latest is a singleton */
Latest := (X : X.ts = MAX[History.ts], X ∈ History)
SEND(READ_RESPONSE, S, Latest)

RECEIVE_READ_PREVIOUS(ts) :
PreHistory := {X : X.ts < ts, X ∈ History}
/* Note, Latest is a singleton */
Latest := (X : X.ts = MAX[PreHistory.ts], X ∈ PreHistory)
SEND(READ_RESPONSE, S, Latest)
```

Figure 1: **Pseudo-code for storage-node S.**

4.1 Storage-node design

Storage-nodes expose the same interface, regardless of the protocol member being employed—write and read requests for all protocol members are serviced identically. Each write request a storage-node executes creates a new version of the data-fragment (indexed by its logical timestamp) at the storage-node (i.e., the storage-node performs comprehensive versioning).

All stored data is initialized to $\perp$ at time $\mathbf{0}$, and has a cross checksum of $\perp$. The zero time, $\mathbf{0}$, and the null value, $\perp$, are well known values which the clients understand.

The storage-node pseudo-code is shown in Figure 1. The *History* which contains the version history for the data-item is kept in stable storage such that it persists during a crash and subsequent recovery. Storage-nodes validate write requests before executing them (to protect against Byzantine clients). This is performed by the function `VALIDATE` called by `RECEIVE_WRITE_REQUEST`. The value returned by `RECEIVE_READ_LATEST` and `RECEIVE_READ_PREVIOUS`, *Latest*, is guaranteed to be unique, since timestamps are unique (i.e., two distinct write operations cannot have the same timestamp).

4.2 Client design

Clients do most of the work, including the execution of the consistency protocol and the encoding and decoding of data-items. The client module provides a block-level interface for reading and writing to higher-level software.

4.2.1 Write operation

```
WRITE(Data) :
 1: /* Encode data, construct timestamp, and write data-fragments */
 2: Time := READ_TIMESTAMP()
 3: {D_1, ..., D_N} := ENCODE(Data)
 4: CC := MAKE_CROSS_CHECKSUM({D_1, ..., D_N})
 5: ts := MAKE_TIMESTAMP(Time, CC)
 6: DO_WRITE({D_1, ..., D_N}, ts, CC)

READ_TIMESTAMP() :
 1: ResponseSet := ∅
 2: repeat
 3:    for all S_i ∈ {S_1, ..., S_N} \ ResponseSet.S do
 4:       SEND(S_i, TIME_REQUEST)
 5:    end for
 6:    if (POLL_FOR_RESPONSE() = TRUE) then
 7:       ⟨S, ts⟩ := RECEIVE_TIME_RESPONSE()
 8:       if (S ∉ ResponseSet.S) then
 9:          ResponseSet := ResponseSet ∪ ⟨S, ts⟩
10:       end if
11:    end if
12: until (|ResponseSet| = N − t)
13: return (MAX[ResponseSet.ts.Time] + 1)

MAKE_CROSS_CHECKSUM({D_1, ..., D_N}) :
 1: for all D_i ∈ {D_1, ..., D_N} do
 2:    H_i := HASH(D_i)
 3: end for
 4: CC := H_1 | ... | H_N
 5: return (CC)

MAKE_TIMESTAMP(Time, CC) :
 1: ts.Time := Time
 2: ts.Verifier := HASH(CC)
 3: return (ts)

DO_WRITE({D_1, ..., D_N}, ts, CC) :
 1: ResponseSet := ∅
 2: repeat
 3:    for all S_i ∈ {S_1, ..., S_N} \ ResponseSet.S do
 4:       SEND(S_i, WRITE_REQUEST, ts, D_i, CC)
 5:    end for
 6:    if (POLL_FOR_RESPONSE() = TRUE) then
 7:       ⟨S⟩ := RECEIVE_WRITE_RESPONSE()
 8:       if (S ∉ ResponseSet.S) then
 9:          ResponseSet := ResponseSet ∪ ⟨S⟩
10:       end if
11:    end if
12: until (|ResponseSet| = N − t)
```

Figure 2: **Client-side write operation pseudo-code.**

The write operation pseudo-code is shown in Figure 2. The `WRITE` operation consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes.

First, a timestamp greater than, or equal to, that of the latest complete write is determined by the `READ_TIMESTAMP` function on line 2 of `WRITE`. Responses are collected, and the highest timestamp is identified, incremented, and returned.

In each iteration of the loop which checks the variable *ResponseSet* in `READ_TIMESTAMP`, additional `TIME_REQUEST` messages are sent to those storage-nodes from which no response has yet been received. Under the crash-recovery failure model, the client must repeatedly send requests until sufficient responses are received to implement a reliable channel. During each iteration of the loop, the client polls to determine if any responses have been received. Only a single response from each storage-node is added to the *ResponseSet*. Once $N-t$ responses are collected, the function returns. Remember, there are at least $N-t$

```
READ(Repair) :
ReadResponseSet := DO_READ(READ_LATEST, ⊥)
loop
    ⟨CandidateSet, ts_candidate⟩ := CHOOSE_CANDIDATE(ReadResponseSet)
    if (|CandidateSet| ≥ COMPLETE) then
        /∗ Complete candidate: return value ∗/
        if (VALIDATE_CANDIDATE_SET(CandidateSet)) then
            Data := DECODE(CandidateSet)
            return (⟨ts_candidate, Data⟩)
        end if
    else if (|CandidateSet| ≥ INCOMPLETE) then
        /∗ Unclassifiable candidate found: repair or abort ∗/
        if (Repair = TRUE) then
            if (VALIDATE_CANDIDATE_SET(CandidateSet) then
                {D_1, ..., D_N} := GENERATE_FRAGMENTS(CandidateSet)
                DO_WRITE({D_1, ..., D_N}, ts_candidate, CC_valid)
                Data := DECODE({D_1, ..., D_N})
                return (⟨ts_candidate, Data⟩)
            end if
        else
            return (ABORT)
        end if
    end if
    /∗ Incomplete candidate or validation failed: loop again ∗/
    ReadResponseSet := DO_READ(READ_PREVIOUS, ts_candidate)
end loop

DO_READ(READ_COMMAND, ts) :
ResponseSet := ∅
repeat
    for all S_i ∈ {S_1, ..., S_N} \ ResponseSet.S do
        SEND(S_i, READ_COMMAND, ts)
    end for
    if (POLL_FOR_RESPONSE() = TRUE) then
        ⟨S, Response⟩ := RECEIVE_READ_RESPONSE()
        if (VALIDATE(Response.D, Response.CC, Response.ts, S) = TRUE) then
            if (READ_COMMAND ≠ READ_PREVIOUS) OR Response.ts < ts then
                if (S ∉ ResponseSet.S) then
                    ResponseSet := ResponseSet ∪ ⟨S, Response⟩
                end if
            end if
        end if
    end if
until (|ResponseSet| = N − t)
return (ResponseSet)

VALIDATE(D, CC, ts, S) :
if ((HASH(CC) ≠ ts.Verifier) OR (HASH(D) ≠ CC[S])) then
    return (FALSE)
end if
return (TRUE)

VALIDATE_CANDIDATE_SET(CandidateSet)
if (ByzantineClients = TRUE) then
    {D_1, ..., D_N} := GENERATE_FRAGMENTS(CandidateSet)
    CC_valid := MAKE_CROSS_CHECKSUM({D_1, ..., D_N})
    if (CC_valid = CandidateSet.CC) then
        return (TRUE)
    else
        return (FALSE)
    end if
end if
return (TRUE)
```

Figure 3: **Client-side read operation pseudo-code.**

good storage-nodes that, eventually, will be up.

Next, the `ENCODE` function, on line 3, encodes the data-item into N data-fragments. Hashes of the data-fragments are used to construct a cross checksum (line 4). The function `MAKE_TIMESTAMP`, called on line 5, generates a logical timestamp for the write operation by combining the hash of the cross checksum and the time determined by `READ_TIMESTAMP`.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. The write operation returns to the issuing client once enough `WRITE_RESPONSE` replies are received (line 12 of `DO_WRITE`).

4.2.2 Read operation

The pseudo-code for the read operation is shown in Figure 3. The read operation iteratively identifies and classifies candidates until either a complete or repairable candidate is found or the operation aborts due to insufficient information (only non-repair members can abort). Before a repairable or complete candidate is returned, the read operation validates its correctness.

The read operation begins by issuing `READ_LATEST` requests to all storage-nodes (via the `DO_READ` function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the highest timestamp it has executed. The integrity of each response is individually validated by the `VALIDATE` function, line 8 of `DO_READ`. This function checks the cross checksum against the validating timestamp and the data-fragment against the appropriate hash in the cross checksum. Since correct storage-nodes perform the same validation before executing write requests, only responses from Byzantine storage-nodes can fail the reader's validation. A second type of validation is performed on read responses on line 9. For responses to `READ_PREVIOUS` commands, the logical timestamp is checked to ensure it is strictly less than the timestamp specified in the command. This check ensures that improper responses from Byzantine storage-nodes are not included in the response set. The read operation does not "count" detectably Byzantine responses towards the $N-t$ total responses collected for the response set. Since $N-t$ storage-nodes are good (by assumption), and the Byzantine storage-node is not good, this action is safe.

After sufficient responses have been received, a candidate for classification is chosen. The function `CHOOSE_CANDIDATE`, on line 3 of `READ`, determines the candidate timestamp, denoted $ts_{\text{candidate}}$, which is the greatest timestamp in the response set. All data-fragments that share $ts_{\text{candidate}}$ are identified and returned as the candidate set. At this point, the candidate set contains a set of data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as either complete, unclassifiable (repairable), or incomplete. If the candidate is classified as incomplete, a `READ_PREVIOUS` message is sent to each storage-node with the candidate timestamp. Candidate classification begins again with the new response set.

If the candidate is classified as complete or repairable, the candidate set is constrained to contain sufficient data-fragments (see Section 5) to decode the original data-item. At this point the candidate is validated. This is done through the `VALIDATE_CANDIDATE_SET` call on line 6 (for complete candidates) or line 13 (for repairable candidates) of `READ`.

For family members that do not tolerate Byzantine clients, this call is a no-op returning `TRUE`. Otherwise, the candidate set is used to generate the full set of data-fragments, as shown in line 2 of `VALIDATE_CANDIDATE_SET`. A validated cross checksum, CC_{valid}, is then computed from the newly generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 4 of `VALIDATE_CANDIDATE_SET`). If the check fails, the candidate was written by a Byzantine client; the candidate is reclassified as incomplete, and the read operation continues. If the check succeeds, the candidate was written correctly and the read enters its final phase. Note that this check either succeeds or fails for all correct clients, regardless of which storage-nodes are represented within the candidate set.

If necessary and allowed, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 15 of READ). Storage-nodes not currently hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it.

Finally, the function DECODE recovers the data-item from any m data-fragments. The read operation returns a logical timestamp, data-item value pair. However, the client likely only makes use of the data-item value.

5 Protocol constraints

To ensure the desired safety and liveness properties are achieved, a number of constraints must hold. For each member protocol, N and m are constrained with regard to b and t (from the hybrid model of storage-node failure). N is the number of storage-nodes in the system and m is the "decode" parameter of an m-of-n erasure code (note, n always equals N in our system).

We develop the protocol constraints under the asynchronous timing model. Synchronous protocol members are considered in Section 8.

5.1 Write operation definitions

We introduce the symbol Q_C in the definition of a complete write operation.

COMPLETE WRITE OPERATION: A write operation is defined to be complete once a total of Q_C benign storage-nodes have executed the write.

Clearly, for a write operation to be durable,

$$t < Q_C. \tag{1}$$

5.2 Read classification rules

Recall that the *candidate* is the data-item version, returned by a read request, with the greatest logical timestamp. The set of read responses that share the candidate's timestamp are the *candidate set*.

To classify a candidate as complete, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|CandidateSet| - b \geq Q_C, \text{ so } \texttt{COMPLETE} = Q_C + b. \tag{2}$$

To classify a candidate as incomplete, the candidate must be incomplete (i.e., fewer than Q_C benign storage-nodes have executed the write). We consider a rule for classifying incomplete writes that takes advantage of $N - t$ responses from storage-nodes. In the crash-recovery model, eventually, a client is guaranteed to receive this many responses—even though, there may be periods during which more than t storage-nodes are crashed. Moreover, in an asynchronous timing model, a client cannot expect more than this many responses, since up to t storage-nodes may never recover. The general rule for classifying a candidate incomplete is,

$$|CandidateSet| + t < Q_C, \text{ so } \texttt{INCOMPLETE} = Q_C - t. \tag{3}$$

Basically, each storage-node that does not respond must be assumed to be benign, and to have executed the write operation.

There are candidates that cannot be classified as complete or incomplete. These candidates are termed unclassifiable/repairable. Repairable protocol family members initiate repair of such candidates, thus completing them. Non-repair protocol family members abort upon encountering unclassifiable candidates.

Protocol	Repairable	Non-repair
N	$2t+2b+1 \leq N$	$3t+3b+1 \leq N$
$\mathbf{Q_C}$	$t+b+1 \leq Q_C$ $Q_C \leq N-t-b$	$t+b+1 \leq Q_C$ $Q_C \leq N-2t-2b$
m	$1 \leq m \leq Q_C - t$	$1 \leq m \leq Q_C + b$

Table 2: **Protocol family constraint summary**

5.3 Protocol properties

This section develops two sets of constraints: one for repairable and one for non-repair protocol members. These constraints hold for asynchronous protocol members under a hybrid crash-recovery–Byzantine model of storage-node failure with Byzantine clients. The constraints do not change if clients only crash. A summary of the constraints for the protocol family is presented in Table 2. Constraints for each protocol family member are derived to satisfy a number of desired properties:

WRITE COMPLETION: This property ensures that write operations by correct clients can complete.

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: This property ensures that colluding Byzantine storage-nodes are unable to fabricate a candidate that a correct client deems unclassifiable/repairable or complete.

CLASSIFIABLE COMPLETE CANDIDATES: This property is only relevant for non-repair protocol members; it ensures that Byzantine storage-nodes cannot make all read operations abort. Consider an isolated correct client that performs a write operation and then a read operation (i.e., the client does not fail and there is no concurrency). This property ensures that the read operation will return the value written by the write operation regardless of storage-node failures.

DECODABLE CANDIDATES: m must be constrained so that complete candidates can be decoded. Moreover, m must be constrained further for repairable protocol members so that repairable candidates can be decoded.

5.4 Repairable constraints

WRITE COMPLETION: There must be sufficient good storage-nodes in the system for a write operation by a correct client to complete. A client must terminate after it receives $N-t$ responses. As well, up to b responses may be Byzantine. Thus, for the write operation to complete (i.e., for Q_C benign storage-nodes to execute write requests),

$$Q_C \leq N-t-b. \tag{4}$$

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: To ensure that Byzantine storage-nodes cannot fabricate an unclassifiable/repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting $|CandidateSet| = b$ into (3),

$$b+t < Q_C. \tag{5}$$

DECODABLE REPAIRABLE CANDIDATES: Any repairable candidate must be decodable. The classification rule for incomplete candidates (cf. (3)) gives the upper bound on m, since a candidate that is not incomplete must be repairable:

$$1 \leq m \leq Q_C - t. \tag{6}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_C + t + b &\leq N; \\ t + b + 1 \leq Q_C &\leq N - t - b; \\ 1 \leq m &\leq Q_C - t. \end{aligned}$$

5.5 Non-repair constraints

Some of the repairable constraints apply to non-repair protocol members. Both the write completion and real unclassifiable/repairable candidates constraints hold (constraints (4) and (5), respectively). However, the write completion constraint is superceded by the classifiable complete candidates constraint.

CLASSIFIABLE COMPLETE CANDIDATES: For this property to hold, a read operation must observe at least $Q_C + b$ responses from benign storage-nodes—sufficient responses to classify the candidate as complete (cf. (2)). A write operation by a correct client may only complete at $N - t$ storage-nodes (due to asynchrony or benign crashes); a subsequent read operation may not observe responses from t benign storage-nodes (again, due to asynchrony or benign crashes). These sets of t storage-nodes that do not respond to the write operation and subsequent read operation could be disjoint sets (since storage-nodes can crash, then recover, or because of asynchrony). Further, b observed responses may be Byzantine. So,

$$\begin{aligned} Q_C + b &\leq ((N - t) - t) - b, \\ Q_C &\leq N - 2t - 2b. \end{aligned} \tag{7}$$

DECODABLE COMPLETE CANDIDATES: A candidate classified as complete, (2), must be decodable:

$$1 \leq m \leq Q_C + b. \tag{8}$$

The upper bound on m is greater than Q_C, even though Q_C defines a complete write operation. The counter-intuitive upper bound on m follows from the fact that a write operation that is complete, may become unclassifiable due to storage-node failures. Given this, only when a write operation is complete and classifiable as such, need it be decodable. However, there is some practical value to constraining $m \leq Q_C$. In a system with failed storage-nodes, a system administrator could make the judgement call that a write operation is complete—so long as $m \leq Q_C$ the system administrator could then force the reconstruction of the data.

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_C + 2t + 2b &\leq N; \\ t + b + 1 \leq Q_C &\leq N - 2t - 2b; \\ 1 \leq m &\leq Q_C + b. \end{aligned}$$

6 Proof of safety

This section presents a proof that our protocol implements linearizability [12] as adapted appropriately for a fault model admitting operations by Byzantine clients.

6.1 Safety guarantees

Intuitively, linearizability requires that each read operation return a value consistent with some execution in which each read and write is performed at a distinct point in time between when the client invokes the operation and when the operation returns. The adaptations necessary to reasonably interpret linearizability in our context arise from the fact that Byzantine clients need not follow the read and write protocols and that read operations may abort in non-repair member protocols. We consider four distinct safety guarantees:

Linearizability. Repairable protocol members with crash-only clients achieve linearizability as originally defined by Herlihy and Wing [12].

Byzantine-operation linearizability. Read operations by Byzantine clients are excluded from the set of linearizable operations. Write operations are only included if they are well-formed (i.e., if they are single-valued as in Section 3).

Write operations by Byzantine clients do not have a well-defined start time. Such operations are concurrent to all operations that begin before they complete and to all operations that are also performed by Byzantine clients. A Byzantine client can write "back in time" by using a lower logical timestamp than a benign client would have used. Since write operations by Byzantine clients are concurrent to all operations that started before it completed, they can be linearized just prior to some concurrent write operation (if there is one). Such a linearization ensures that the Byzantine "back in time" write operation has no effect since the value written is never returned by a read operation.

In summary, there are two types of Byzantine write operations that are of concern: writes that are not well-formed and "back in time" writes. In the case that the Byzantine write operation is not well-formed, read operations by benign clients exclude it from the set of linearized operations. In the case that the Byzantine write operation is "back in time", the protocol family achieves something similar, in that such Byzantine write operations are linearized so that they have no effect.

Linearizability with read aborts. Non-repair protocol members allow reads to abort due to insufficient classification information: aborted reads are excluded from the set of linearizable operations. Such members achieve "linearizability with read aborts", which is similar to Pierce's "pseudo-atomic consistency" [22]. That is, the set of all write operations and all complete read operations is linearizable.

Byzantine-operation linearizability with read aborts. For non-repair protocol members that tolerate Byzantine clients, the safety property is the combination of Byzantine-operation linearizability and linearizability with read aborts.

6.2 Proof

Because return values of reads by Byzantine clients obviously need not comply with any correctness criteria, we disregard read operations by Byzantine clients in reasoning about linearizability, and define the duration of reads only for those executed by benign clients only.

DEFINITION 1 A read operation executed by a benign client *begins* when the client invokes `READ` locally. A read operation executed by a benign client *completes* when this invocation returns $\langle timestamp, value \rangle$. A read operation by a benign client that crashes before the read completes, does not complete.

Before defining the duration of write operations, it is necessary to define what it means for a storage-node to *accept* and then *execute* a write request.

DEFINITION 2 Storage-node S, *accepts* a write request with data-fragment D, cross checksum CC, and timestamp ts upon successful return of the function $\texttt{VALIDATE}(ts, D, CC)$ at the storage-node.

DEFINITION 3 Storage-node S, *executes* a write request once the write request is accepted. An executed write request is stored in stable storage.

It is not well defined when a write operation by a Byzantine client begins. Therefore, we settle for merely a definition of when writes by Byzantine clients complete.

DEFINITION 4 A write operation with timestamp *ts completes* once Q_C benign storage-nodes have executed write requests with timestamp *ts*.

In fact, Definition 4 applies to write operations by benign clients as well as "write operations" by Byzantine clients. In this section, we use the label w_{ts} as a shorthand for the write operation with timestamp *ts*. In contrast to Definition 4, Definition 5 applies only to write operations by benign clients.

DEFINITION 5 w_{ts} *begins* when a benign client invokes the `WRITE` operation locally that issues a write request bearing timestamp *ts*.

LEMMA 1 *Let c_1 and c_2 be benign clients. If c_1 performs a read operation that returns $\langle ts_1, v_1 \rangle$, c_2 performs a read operation that returns $\langle ts_2, v_2 \rangle$, and $ts_1 = ts_2$, then $v_1 = v_2$.*

Proof: Since $ts_1 = ts_2$, each read operation considers the same verifier. Since each read operation considers the same verifier, each read operation considers the same cross checksum (remember, a collision resistant hash function is employed). A read operation does not return a value unless the cross checksum is valid and there are more than b read responses with the timestamp (since only candidates classified as repairable or complete are considered). Thus, only a set of data-fragments resulting from the erasure-coding of the same data-item that are issued as write requests with the same timestamp can validate a cross checksum. As such, v_1 and v_2 must be the same. □

Let v_{ts} denote the value written by w_{ts} which, by Lemma 1, is well-defined. We use r_{ts} to denote a read operation by a benign client that returns $\langle ts, v_{ts} \rangle$.

DEFINITION 6 Let o_1 denote an operation that completes (a read operation by a benign client, or a write operation), and let o_2 denote an operation that begins (a read or write by a benign client). o_1 *precedes* o_2 if o_1 completes before o_2 begins. The precedence relation is written as $o_1 \rightarrow o_2$. Operation o_2 is said to follow, or to be subsequent to, operation o_1.

LEMMA 2 *If $w_{ts} \rightarrow w_{ts'}$, then $ts < ts'$.*

Proof: A complete write operation executes at at least Q_C benign storage-nodes (cf. Definition 4). Since $w_{ts} \rightarrow w_{ts'}$, the `READ_TIMESTAMP` function for w_{ts} collects $N - t$ `TIME_RESPONSE` messages, and so $w_{ts'}$ observes at least $b+1$ `TIME_RESPONSE` messages from benign storage-nodes that executed w_{ts} (remember, $t + b < Q_C$ for all asynchronous protocol family members). As such, $w_{ts'}$ observes some timestamp greater than or equal to *ts* and constructs ts' to be greater than *ts*. A Byzantine storage-node can return a logical timestamp greater than that of the preceding write operation; however, this still advances logical time and Lemma 2 holds. □

OBSERVATION 1 Timestamp order is a total order on write operations. The timestamps of write operations by benign clients respect the precedence order among writes.

LEMMA 3 *If some read operation by a benign client returns $\langle ts, v_{ts} \rangle$, with $v_{ts} \neq \perp$, then w_{ts} is complete.*

Proof: For a read operation to return value v_{ts}, the value must have been observed at at least $Q_C + b$ storage-nodes (given the complete classification rule for candidate sets). Since, at most b storage-nodes are Byzantine, the write operation w_{ts} has been executed by at least Q_C benign storage-nodes. By definition, w_{ts} is complete. □

OBSERVATION 2 The read operation from Lemma 3 could have performed repair before returning. In a repairable protocol member, a candidate that is neither classifiable as incomplete or complete is repaired. Once repaired, the candidate is complete.

DEFINITION 7 w_{ts} is *well-formed* if *ts.Verifier* equals the hash of cross checksum CC, and for all $i \in \{1,\ldots,N\}$, hash $CC[i]$ of the cross checksum equals the hash of data-fragment i that results from the erasure-encoding of v_{ts}.

LEMMA 4 *If w_{ts} is well-formed, and if $w_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$.*

Proof: Since w_{ts} is well-formed it can be returned by a read operation. By Lemma 3, read operations only return values from complete write operations. As such, $r_{ts'}$ must either return the value with timestamp ts or a value with a greater timestamp. Therefore, $ts \leq ts'$. □

OBSERVATION 3 It follows from Lemma 4 that for any read r_{ts}, either $w_{ts} \rightarrow r_{ts}$ and w_{ts} is the latest complete write that precedes r_{ts}, or $w_{ts} \not\rightarrow r_{ts}$ and $r_{ts} \not\rightarrow w_{ts}$ (i.e., w_{ts} and r_{ts} are concurrent).

OBSERVATION 4 It also follows from Lemmas 3 and 4 that if $r_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$. As such, there is a partial order $\prec$ on read operations by benign clients defined by the timestamps associated with the values returned (i.e., of the write operations read). More formally, $r_{ts} \prec r_{ts'} \Longleftrightarrow ts < ts'$.

Since Lemma 2 ensures a total order on write operations, ordering reads according to the timestamps of the write operations whose values they return yields a partial order on read operations. Lemma 4 ensures that this partial order is consistent with precedence among reads. Therefore, any way of extending this partial order to a total order yields an ordering of reads that is consistent with precedence among reads. Thus, Lemmas 2 and 4 guarantee that this totally ordered set of operations is consistent with precedence. This implies the natural extension of linearizability to our fault model (i.e., ignoring reads by Byzantine clients and the begin time of writes by Byzantine clients); in particular, it implies linearizability as originally defined by Herlihy [12] for repairable protocol family members if all clients are benign.

OBSERVATION 5 Note, Lemma 4 does not address reads that abort (it only addresses reads that return a value). Read operations that abort are excluded from the set of operations that are linearized.

7 Proof of liveness

This section presents the proof of the liveness properties of protocol members.

7.1 Liveness guarantees

There are two distinct liveness guarantees: wait-freedom and single-client wait-freedom. These guarantees hold so long as the storage capacity on storage-nodes is not exhausted.

Wait-freedom. Wait-freedom is a desirable liveness property [10]. Informally, achieving wait-freedom means that each client can complete its operations in finitely many steps regardless of the actions performed or failures experienced by other clients. For a formal definitions see [10].

Single-client wait-freedom. In non-repair protocol members, wait-freedom is not achieved. This is because read operations may abort due to concurrency or the failure of other clients. The strongest statement about the liveness of non-repair member protocols is that a single correct client is wait-free. I.e., in a system which is comprised of a single correct client that performs operations sequentially, all operations are wait-free. The write operations of all protocol members are wait-free; it is only the read operation for which the weaker single-client wait-freedom is required.

Unbounded storage capacity. In the proof of liveness for read operations, we assume that storage-nodes have unbounded storage capacity (i.e., that the entire version history back to the initial value $\perp$ at time $\mathbf{0}$

is available at each storage-node). To prevent capacity exhaustion, some garbage collection mechanism is required. Garbage collection reduces the liveness of read operations. A read operation that is concurrent to write operations and to garbage collection may not observe a complete candidate. The read operation can observe a series of incomplete candidates that complete and are garbage collected within the duration of the read operation. In such a situation, the read operation would observe $\perp$ at some timestamp other than **0** from storage-nodes, indicating that the client has "skipped" over a complete write operation. The read operation then must be retried. The implementation details of garbage collection and its impact on liveness properties is given in [8].

7.2 Proof

All liveness properties hinge on the following lemma.

LEMMA 5 *All operations eventually receive at least $N-t$ responses.*

Proof: In the crash-recovery model, there are at least $N-t$ good storage-nodes (i.e., storage-nodes that are always-up or eventually-up). By definition, eventually, all good storage-nodes will be up. Since all requests to storage-nodes, from clients, are retried until $N-t$ responses are received, eventually, $N-t$ responses will be received (see `READ_TIMESTAMP`, `DO_WRITE`, and `DO_READ`). $\square$

OBSERVATION 6 It is possible for progress to be made throughout the duration of a run, not just once all good storage-nodes are up. Lemma 5 guarantees that eventually $N-t$ responses will be received. During any period in which $N-t$ storage-nodes are up, operations may receive $N-t$ responses and thus complete. In fact, responses can be collected, over time, from $N-t$ storage-nodes, during a period in which fewer than $N-t$ storage-nodes are ever up (but during which some storage-nodes crash and some recover).

7.2.1 Asynchronous repairable

The asynchronous repairable protocol member provides a strong liveness property, namely wait-freedom [10, 14]. Informally, each operation by a correct client completes with certainty, even if all other clients fail, provided that at most b servers suffer Byzantine failures and no more than t servers are not good.

LEMMA 6 *A write operation by a correct client completes.*

Proof: A write operation by a correct client waits for $N-t$ responses from storage-nodes before returning. By Lemma 5, $N-t$ responses can always be collected. Since, $Q_C \leq N-t-b$ (cf. (4) in Section 5) for repairable protocol members, then $N-t \geq Q_C + b$. Since at most b storage-nodes are Byzantine, then at least Q_C benign storage-nodes execute write requests, which completes the write operation. $\square$

LEMMA 7 *A read operation by a correct client completes.*

Proof: Given $N-t$ `READ_RESPONSE` messages, a read operation classifies a candidate as complete, repairable, or incomplete. The read completes if a candidate is classified as complete. As well, the read completes if a candidate is repairable. Repair is initiated for repairable candidates—repair performs a write operation, which by Lemma 6 completes—which lets the read operation complete. In the case of an incomplete, the read operation traverses the version history backwards, until a complete or repairable candidate is discovered. Traversal of the version history terminates if $\perp$ at logical time **0** is encountered at Q_C storage-nodes. $\square$

7.2.2 Asynchronous non-repair

Like the asynchronous repairable member, write operations of the asynchronous non-repair member are wait-free. Indeed, Lemma 6 holds, since $Q_C \leq N-2t-2b \leq N-t-b$ for the asynchronous non-repair member (cf. (7) in Section 5).

Read operations of the asynchronous non-repair member may abort (i.e., return $\perp$ at some time greater than **0**). However, we can make a limited statement about the liveness of read operations.

LEMMA 8 *In a system comprised of a single correct client, all read operations complete.*

Proof: Write operations by the correct client are executed by at least $N-t-b$ benign storage-nodes. Since $Q_C \leq N-2t-2b$, then at least Q_C+b of the benign storage-nodes that execute the write operation are in the candidate set of a subsequent read operation. A candidate set of size Q_C+b is classified as complete. Therefore, read operations complete. □

8 Synchronous protocol family members

In this section we consider the constraints on N, Q_C, and m for synchronous members of the protocol family. We consider the constraints under three related failure models: crash-recovery, omission, and fail-stop. The crash-recovery failure model is a strict generalization of the omission and fail-stop failure models. The omission failure model is a strict generalization of the fail-stop failure model. Synchronous protocol members differ from asynchronous in that there are distinct constraints for each failure model. The lower bound on N decreases as the storage-node failures tolerated become less general. We maintain a hybrid failure model, in that b storage-nodes may be Byzantine, and $t \geq b$ storage-nodes may have omissions (or, may fail-stop).

8.1 Crash-recovery

In a synchronous protocol member, it is possible to wait for responses from all storage-nodes (where `TIMEOUT` may be the response). Thus, if more than $N-t$ storage-nodes are up, the read classification rule has more information with which to classify incomplete candidates. To classify a candidate as incomplete, the candidate must be incomplete (i.e., fewer than Q_C benign storage-nodes have executed the write). Let f be the number of storage-nodes that have timed out. Since operations retry requests until $N-t$ responses are received, $0 \leq f \leq t$. In synchronous members, the rule for classifying a candidate incomplete is,

$$|CandidateSet| + f < Q_C, \text{ so } \mathtt{INCOMPLETE} = Q_C - f, \tag{9}$$

Basically, each storage-node that has timed out must be assumed to be benign, and to have executed the write operation.

In the case that t storage-nodes timeout in a system, then (9) is identical to (3). However, if fewer than t storage-nodes timeout, then (9) has more information with which to classify candidates as incomplete.

The ability to better classify incomplete candidates is the major difference between asynchronous and synchronous protocol members in the crash-recovery model. Specifically, the constraints on N, Q_C, and m, listed in Table 2 apply to the synchronous crash-recovery protocol members. As well, the safety proof given in Section 6 and liveness proofs given in Section 7 applies.

Protocol	Omission repairable	Omission non-repair	Fail-stop repairable	Fail-stop non-repair
N	$2t+1 \leq N$	$2t+b+1 \leq N$	$t+b+1 \leq N$	$t+2b+1 \leq N$
$\mathbf{Q}_{\mathrm{C}}$	$t+1 \leq Q_{\mathrm{C}}$ $Q_{\mathrm{C}} \leq N-t$	$t+1 \leq Q_{\mathrm{C}}$ $Q_{\mathrm{C}} \leq N-t-b$	$t+1 \leq Q_{\mathrm{C}}^{\mathrm{FS}}$ $Q_{\mathrm{C}}^{\mathrm{FS}} \leq N-b$	$t+1 \leq Q_{\mathrm{C}}^{\mathrm{FS}}$ $Q_{\mathrm{C}}^{\mathrm{FS}} \leq N-2b$
m	$1 \leq m \leq Q_{\mathrm{C}}-t$	$1 \leq m \leq Q_{\mathrm{C}}$	$1 \leq m \leq Q_{\mathrm{C}}^{\mathrm{FS}}-t$	$1 \leq m \leq Q_{\mathrm{C}}^{\mathrm{FS}}-t+b$

Table 3: **Synchronous protocol family members constraint summary**

```
DO_WRITE({D_1,...,D_N}, ts, CC) :
1: for all S_i ∈ {S_1,...,S_N} do
2:    SEND(S_i, WRITE_REQUEST, ts, D_i, CC)
3: end for
4: ResponseSet := ∅
5: repeat
6:    ResponseSet := ResponseSet ∪ RECEIVE_WRITE_RESPONSE(S)
7: until (|ResponseSet| = N OR TIMEOUT)
```

Figure 4: **Synchronous implementation of DO_WRITE with reliable channels.**

8.2 Less general failure models

A distinguishing feature of the omission and fail-stop failure models, compared to the crash-recovery model, is that the locations of failures are fixed. That is, $N-t$ storage-nodes are always-up. There are no storage-nodes that are eventually-up—no more than t storage-nodes may experience failures of any kind.

The constraints on N, m, and Q_{C} for synchronous protocol members under the omission and fail-stop failure models are given in Table 3. Notice that the definition of Q_{C} is modified for the fail-stop failure model (thus, the term $Q_{\mathrm{C}}^{\mathrm{FS}}$).

8.2.1 Reliable channels

Under the omission and fail-stop failure models, we assume reliable channels. Under the crash-recovery model, we implement reliable channels by repeatedly sending requests to sets of storage-nodes until sufficient responses are received. It is easier to think about the synchronous omission and fail-stop protocol members by assuming reliable channels.

The assumption of reliable channels affects the implementation of the functions `READ_TIMESTAMP`, `DO_WRITE`, and `DO_READ`. The change of implementation is trivial. We illustrate the modified implementation with the function `DO_WRITE` in Figure 4. The functions `READ_TIMESTAMP` and `DO_READ` are similarly modified.

Since the channels are assumed to be reliable, messages to the storage-nodes are sent just once (cf. line 2). Responses are collected until a total of N responses are collected, or until the `TIMEOUT` for the synchronous system is reached (cf. line 7). If the loop exits because `TIMEOUT` is reached, by assumption, the response set has at least $N-t$ members. Strictly speaking, the second half of the function which collects responses to the write requests is unnecessary. It may be useful for the client to know which storage-nodes acknowledge the write requests, and which have timed out; however, because of the reliable channels, `DO_WRITE` could return after line 3.

8.3 Omission failure model

A storage-node that experiences an omission failure either does not receive, or does not send a message [21]. A client that "observes" an omission, in that the storage-node did not reply within the synchronous bound of channel delay and processing delay, receives a timeout. Given that no more than t storage-nodes may fail, a client is guaranteed to receive N responses, no more than t of which are timeouts. If a client receives a timeout from some storage-node, then some other client, in a subsequent request, may receive a response from the storage-node. This is the nature of omission failures.

Under the omission failure model, the definition of a complete write operation is the same as in the crash-recovery model. However, the classification rule for a complete candidate is modified to take advantage of observed failures. Defining f, as above, as the number of timeouts received, if $f > t-b$, then there are at most $t-f$ Byzantine storage-nodes in the candidate set. For example, if $f = t$, then all responses received, that are not timeouts, are from correct storage-nodes. To reason about this ability in the constraints, we introduce $\hat{b}$, which is defined as follows:

$$\hat{b} = \begin{cases} b & \text{if } f \le t-b; \\ t-f & \text{if } f > t-b. \end{cases} \tag{10}$$

In a synchronous model with omission failures, the complete classification rule is,

$$|CandidateSet| - \hat{b} \ge Q_{\mathrm{C}}, \text{ so } \texttt{COMPLETE} = Q_{\mathrm{C}} + \hat{b}. \tag{11}$$

The incomplete classification rule is the same as in the crash-recovery model,

$$|CandidateSet| + f < Q_{\mathrm{C}}, \text{ so } \texttt{INCOMPLETE} = Q_{\mathrm{C}} - f. \tag{12}$$

The more responses received by a client, the "better" it can classify incomplete candidates, since f is lower.

8.3.1 Repairable protocol member constraints

WRITE COMPLETION: There must be sufficient correct storage-nodes in the system for a write operation to complete. Since only t storage-nodes may fail, and since correct storage-nodes always reply to clients, then

$$Q_{\mathrm{C}} \le N - t. \tag{13}$$

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: No additional constraints are necessary to ensure that Byzantine storage-nodes cannot fabricate repairable candidates. Consider (12), the incomplete classification rule under omission failures in synchronous systems. If b Byzantine storage-nodes fabricate a candidate, then at most $t-b$ storage-nodes timeout. Substituting $f = t-b$ into (12), we have $|CandidateSet| + t - b < Q_{\mathrm{C}}$. Since, $|CandidateSet| \le b$, then this becomes $t < Q_{\mathrm{C}}$ (which is redundant, given (1), the write durability constraint). So long as the candidate set has b or fewer members, it is classified as incomplete; therefore, Byzantine storage-nodes cannot fabricate a repairable candidate.

DECODABLE REPAIRABLE CANDIDATES: If t storage-nodes do not respond, then the classification rule for incomplete, (12), leads to the threshold for repairable candidates,

$$1 \le m \le Q_{\mathrm{C}} - t. \tag{14}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_{\mathrm{C}} + t &\le N; \\ t + 1 \le Q_{\mathrm{C}} &\le N - t; \\ 1 \le m &\le Q_{\mathrm{C}} - t. \end{aligned}$$

8.3.2 Non-repair protocol member constraints

The above real unclassifiable candidate constraint holds for the non-repair protocol member (i.e., $t < Q_{\mathrm{C}}$). The write completion constraint is superceded by the classifiable complete candidates constraint.

CLASSIFIABLE COMPLETE CANDIDATES: For this property to hold, a read operation must observe at least $Q_{\mathrm{C}} + b$ responses from correct storage-nodes—sufficient responses to classify the candidate as complete. Since $\hat{b}$ may equal b in the complete classification rule, (11), and since a write operation may only receive responses from $N - t$ storage-nodes, then

$$\begin{aligned} Q_{\mathrm{C}} + b &\leq N - t, \\ Q_{\mathrm{C}} &\leq N - t - b. \end{aligned} \tag{15}$$

DECODABLE COMPLETE CANDIDATES: A candidate classified as complete, (11), must be decodable. Since, if $f = t$, $\hat{b} = 0$, it is possible to classify something as complete with a candidate set with only Q_{C} members,

$$1 \leq m \leq Q_{\mathrm{C}}. \tag{16}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_{\mathrm{C}} + t + b &\leq N; \\ t + 1 \leq Q_{\mathrm{C}} &\leq N - t - b; \\ 1 \leq m &\leq Q_{\mathrm{C}}. \end{aligned}$$

8.4 Fail-stop failure model

A storage-node that experiences a fail-stop failure, crashes in such a manner that it takes no further action and its failure is detectable [25]. In a synchronous system, this means that clients can detect that a storage-node has failed—although Byzantine storage-nodes can appear fail-stopped to some clients and up to other clients. Under the fail-stop failure model, the definition of a complete write operation is modified from that of the crash-recovery model (and omission model).

COMPLETE WRITE OPERATION: A write operation is defined to be complete once a total of $Q_{\mathrm{C}}^{\mathrm{FS}}$ benign storage-nodes have executed write requests or have fail-stopped.

The change in definition of a complete write operation affects the classification rules for complete and incomplete candidates:

$$|CandidateSet| + f - b \geq Q_{\mathrm{C}}^{\mathrm{FS}}, \text{ so } \texttt{COMPLETE} = Q_{\mathrm{C}}^{\mathrm{FS}} - f + b; \tag{17}$$

$$|CandidateSet| + f < Q_{\mathrm{C}}^{\mathrm{FS}}, \text{ so } \texttt{INCOMPLETE} = Q_{\mathrm{C}}^{\mathrm{FS}} - f. \tag{18}$$

For the complete classification rule, it is assumed that up to b of the responses could be from Byzantine storage-nodes (such storage-nodes could by lying about a value or pretending to be fail-stopped). For the incomplete classification rule, it is assumed that all observed fail-stop responses are from benign storage-nodes.

8.4.1 Repairable protocol member constraints

WRITE COMPLETION: There must be sufficient benign storage-nodes in the system for a write operation to complete. Since only b storage-nodes may be Byzantine, and the remainder are benign,

$$Q_{\mathrm{C}}^{\mathrm{FS}} \leq N - b. \tag{19}$$

REAL UNCLASSIFIABLE/REPAIRABLE CANDIDATES: For reasons similar to the omission failure model, the write durability constraint ($t < Q_{\mathrm{C}}^{\mathrm{FS}}$) is sufficient for ensuring that Byzantine storage-nodes cannot fabricate repairable candidates. Specifically, if b Byzantine storage-nodes fabricate a candidate, then at most $t-b$ storage-nodes fail-stop, so from (18), we have $|CandidateSet| + t - b < Q_{\mathrm{C}}^{\mathrm{FS}}$. And, since $Q_{\mathrm{C}}^{\mathrm{FS}} > t$, we have, $|CandidateSet| \leq b$. Therefore Byzantine storage-nodes cannot fabricate a repairable candidate.

DECODABLE REPAIRABLE CANDIDATES: Given $f = t$ failures, an operation can be classified as repairable with as few as $Q_{\mathrm{C}}^{\mathrm{FS}} - t$ members in the candidate set, thus,

$$1 \leq m \leq Q_{\mathrm{C}}^{\mathrm{FS}} - t. \tag{20}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_{\mathrm{C}}^{\mathrm{FS}} + b &\leq N; \\ t + 1 \leq Q_{\mathrm{C}}^{\mathrm{FS}} &\leq N - b; \\ 1 \leq m &\leq Q_{\mathrm{C}}^{\mathrm{FS}} - t. \end{aligned}$$

8.4.2 Non-repair protocol member constraints

The bound on the above write completion constraint is superceded by the classifiable complete candidates constraint.

CLASSIFIABLE COMPLETE CANDIDATES: For this property to hold, a read operation must observe at least $Q_{\mathrm{C}}^{\mathrm{FS}} - f + b$ responses that match. Considering the case of $f = 0$ leads to the lower bound on N. If $f = 0$, then $Q_{\mathrm{C}}^{\mathrm{FS}} + b$ responses must match. Since up to b storage-nodes can lie, then there must be at least $Q_{\mathrm{C}}^{\mathrm{FS}} + b + b \leq N$ storage-nodes to ensure this property.

$$\begin{aligned} Q_{\mathrm{C}}^{\mathrm{FS}} + b + b &\leq N, \\ Q_{\mathrm{C}}^{\mathrm{FS}} &\leq N - 2b. \end{aligned} \tag{21}$$

DECODABLE COMPLETE CANDIDATES: A candidate classified as complete, (17), must be decodable. Since, f may be as great as t,

$$1 \leq m \leq Q_{\mathrm{C}}^{\mathrm{FS}} - t + b. \tag{22}$$

CONSTRAINT SUMMARY:

$$\begin{aligned} Q_{\mathrm{C}}^{\mathrm{FS}} + 2b &\leq N; \\ t + 1 \leq Q_{\mathrm{C}}^{\mathrm{FS}} &\leq N - 2b; \\ 1 \leq m &\leq Q_{\mathrm{C}}^{\mathrm{FS}} - t + b. \end{aligned}$$

9 Synchronous members with synchronized clocks

Protocols to achieve approximate clock synchronization in today's networks are well known, inexpensive, and widely deployed [20]. In this section, we consider the use of synchronized clocks for synchronous protocol family members. Synchronized clocks and weakening safety allow the write operation to be performed in a single phase, rather than in two phases. Specifically, there is no need to determine the "current" logical time by sending requests to storage-nodes; the local clock time is used. Thus, line 2 of the function `WRITE` in Figure 2 becomes *Time* := `READ_LOCAL_CLOCK()`, and the function `READ_TIMESTAMP` is not needed.

9.1 Clock skew

The use of client clock synchronization introduces clock skew (i.e., clocks are not perfectly synchronized—they are synchronized to within some bound, the skew). Since there may be skew among client clocks, the definition of operation duration must be modified; this weakens the safety guarantee. Specifically, we extend the definition of when a write operation begins, to accommodate that, within the clock skew, write operations may be linearized "out of order".

From Lemma 2, we observed that the timestamp order is a total order on write operations. We define τ to be the maximum skew between the clocks of any two correct clients. For synchronous members with synchronized clocks, we redefine when a benign client begins a write operation (cf. Definition 5):

DEFINITION 8 w_{ts} *begins* τ before a benign client invokes the `WRITE` operation locally that issues a write request bearing timestamp ts.

Consider two clients, A and B, whose clocks differ in that A's clock is τ less than B's clock. It is necessary to extend the begin time of a write operation to accommodate the case when B invokes a write operation less than τ before A invokes a write operation. If B executes write requests at Q_C benign storage-nodes before A invokes its write operation, then B's write operation is complete when A's write operation begins. However, A's write operation will be linearized before B's write operation—it is linearized "out of order".

9.2 Byzantine clients

Using purely logical time, the begin times of write operations by Byzantine clients is undefined. As discussed in Section 6.1, this allows such write operations to be linearized just prior to some other write operation such that no read operation observes them (i.e., such that they have no effect). Write operations by Byzantine clients that are "back-in-time" in synchronous members with synchronized clocks can be treated similarly.

Byzantine clients that write "into the future" require additional logic during the read operation to handle correctly. A correct client must ignore (classify as incomplete) candidates with a timestamp greater than τ in the future relative to its local clock. No correct client can perform a write operation which a subsequent read operation by a correct client observes as being greater than τ in the future relative to its local clock.

If storage-nodes, as well as clients, have synchronized clocks, then storage-nodes can reject write requests that are "into the future" (i.e., greater than τ in the future relative to its local clock). Given storage-nodes with synchronized clocks, no additional client logic is necessary on a read operation; Byzantine clients cannot write "into the future".

Moreover, storage-nodes could reject "back in time" write requests. The bound on transmission delay and processing time is used to define a "back in time" write request. Let ϕ be the maximum delay due to message transmission, sender processing after reading its local clock, and receiver processing before reading its local clock. Thus, a storage-node can reject a "back in time" write request if the timestamp of the request is more than $\tau + \phi$ in the past relative to its local clock. By assumption/definition, only Byzantine clients can issue write requests which correct storage-nodes reject. Note that a storage-node with a clock that is out of synchronization is considered to be Byzantine.

10 Related work

We review work related to the concept of protocol families, access protocols, and survivable storage systems (including quorum systems and erasure-coded systems) in [9]. Here, we focus on work related to the failure models employed and the subsequent safety and liveness properties achieved by members of the protocol family (especially in the context of Byzantine clients).

We note that the protocol family was developed in the context of the PASIS survivable storage system project [28, 9, 7]. Since storage-nodes actually have finite capacity, a garbage collection mechanism is needed. Discussion of some implementation details of garbage collection and its impact on liveness properties is given in [8].

10.1 Failure models

We consider many failure models. We make use of a hybrid failure model for storage-nodes [27]. However, we move beyond a mix of crash and Byzantine failures to allow for a mix of crash-recovery, omission or fail-stop failures and Byzantine failures. The crash-recovery model was introduced by Aguilera, Chen, and Toueg [1]. The omission failure model was introduced by Perry and Toueg in [21]. The fail-stop failure model was introduced by Schlichting and Schneider [24, 25]. The Byzantine failure model was introduced by Lamport, Shostak, and Pease [16]. Backes and Cachin have also considered the "hybridization" of Byzantine faults with a crash-recovery model for reliable broadcast [3].

The protocol family we have developed is not adaptive with regard to the faults tolerated—each family member tolerates a static failure model. This is in clear contrast to work by Chen, Hiltunen, and Schlichting [13, 5] in which (fault-tolerant) systems are developed that gracefully adapt to changes in the execution environment or user requirements by switching the protocols employed. Adaptive techniques for Byzantine quorum systems were developed by Alvisi, Malkhi, Pierce, Reiter, and Wright [2]. The application of adaptive fault thresholds to Byzantine quorum systems could inform future extensions of our protocol family.

10.2 Safety, liveness, and Byzantine clients

To provide reasonable storage semantics a system must guarantee that readers see consistent answers. For shared storage systems, this usually means linearizability [12] of operations. Jayanti refined the notion of wait-freedom to address fault-tolerant shared objects [14].

Although we focus on achieving wait-freedom, we also had to consider what we called single-client wait-freedom (see Section 6.1). It is similar to, but weaker than, obstruction-freedom [11] by Herlihy, Luchango, and Moir. Obstruction-freedom guarantees progress once concurrency subsides. Single-client wait-freedom guarantees progress once concurrency subsides only if all clients are correct (assuming clients retry if a read operation aborts in non-repair protocol members).

In synchronous fail-stop protocol members, the safety and liveness guarantees are not *pure*. Charron-bost, Toug, and Basu identify safety and liveness properties that are not pure in [4]. Such properties can become true because of failures. Our definition of a complete write operation for synchronous fail-stop protocol members is clearly not pure: We include benign storage-nodes that have fail-stopped in the definition of $Q_{\mathrm{C}}^{\mathrm{FS}}$ (see Section 8.4).

Pierce extended linearizability to include the possibility of read aborts: pseudo-regular semantics [22]. If a reader sees either a consistent answer or aborts, it achieves pseudo-regular semantics. Trivial solutions (i.e., readers always aborting) are excluded by identifying specific conditions under which a read operation must return a value. The liveness guarantee of protocol members that allow aborts, linearizability with read aborts, is very similar to Pierce's pseudo-regular semantics.

Write operations in the protocol family by Byzantine clients do not have a well-defined start time and are thus concurrent to all preceding operations. Members of the protocol family linearize writes by Byzantine clients, if they are "back in time", just prior to some other write operation so that they have no effect. Other work does not directly discuss Byzantine clients that write "back in time". Malkhi and

Reiter in [18] use an "echo" protocol to ensure that write operations are well-formed ("justified" in their terminology). The echo protocol requires an additional phase—indeed, the protocol employs three phases: get time, propose (echo), and commit. The benefit of the echo protocol is two-fold. First, it appears that Byzantine clients could be prevented from writing "back in time", since the propose message includes signed get time responses. However the server logic presented in [18] does not directly address "back in time" writes. Indeed, the prior work of Malkhi and Reiter in [17], indicates that "back in time" write operations by Byzantine clients are treated in a manner similar to the protocol family (Section 3.2 of [17] states that each server modifies its value and timestamp only if the timestamp received is greater than the latest timestamp received—i.e., "back in time" writes are treated as if they have no effect). Second, the echo phase ensures that a Byzantine client is sending the same value to each server. Adding an echo phase to the protocol family could allow the begin time of write operations to be defined, but would not achieve the second benefit, since the protocol family allows values to be erasure-coded.

Martin, Alvisi, and Dahlin in [19] use a different approach for dealing with Byzantine clients in the Minimal Byzantine Storage protocol. To deal with Byzantine clients, Martin et al. depart from the traditional quorum communication pattern and allow inter-server communication. Whenever a correct server receives a write request, it stores the value and then broadcasts the value to other servers. Treating the data as the low bits of the timestamp, all correct servers can "agree" on the latest value written, even if a Byzantine client sends different data with the same timestamp to each server. Again, Byzantine clients writing "back in time" is not directly discussed (in terms of its ramifications on linearizability). However, the protocol appears to guarantee a similar property to the protocol family: "back in time" writes have no effect.

Linearizability is not well defined in the context of Byzantine clients. We believe that there may be many useful variations of linearizability with Byzantine clients. Unfortunately, the useful variations may depend on the approach taken by the protocol.

References

[1] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, **13**(2):99–125. Springer-Verlag, 2000.

[2] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K. Reiter, and Rebecca N. Wright. Dynamic byzantine quorum systems. *Dependable Systems and Networks*, pages 283–292. IEEE, 2000.

[3] Michael Backes and Christian Cachin. Reliable broadcast in a computational hybrid model with Byzantine faults, crashes, and recoveries. *Dependable Systems and Networks*, pages 37–46. IEEE, 2003.

[4] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. *International Conference on Concurrency Theory*, pages 552–565, 2000.

[5] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. *International Conference on Distributed Computing Systems* (Phoenix, AZ, 16–19 April 2001), pages 635–643. IEEE, 2001.

[6] Li Gong. Securely replicating authentication services. *International Conference on Distributed Computing Systems* (Newport Beach, CA), pages 85–91. IEEE Computer Society Press, 1989.

[7] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. *International Conference on Dependable Systems and Networks* (Florence, Italy, 28 June – 01 July 2004), 2004.

[8] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Efficient Byzantine-tolerant erasure-coded storage*. Technical report CMU-PDL-03-104. CMU, December 2003.

[9] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *A protocol family for versatile survivable storage infrastructures*. Technical report CMU-PDL-03-103. CMU, December 2003.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, **13**(1):124–149. ACM Press, 1991.

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: double-ended queues as an example. *International Conference on Distributed Computing Systems* (Providence, RI, 19–22 May 2003), pages 522–529. IEEE, 2003.

[12] 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.

[13] Matti A. Hiltunen and Richard D. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, **11**(5):275–285. CRL. Publishing, Sept. 1996.

[14] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, **45**(3):451–500. ACM Press, May 1998.

[15] Hugo Krawczyk. Secret sharing made short. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 22–26 August 1993), pages 136–146. Springer-Verlag, 1994.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[17] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. *IEEE Symposium on Reliable Distributed Networks* (West Lafayette, IN, 20–23 October 1998), 1998.

[18] Dahlia Malkhi and Michael K. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, **12**(2). IEEE, April 2000.

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

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

[21] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, **SE–12**(3):477–482, March 1986.

[22] 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.

[23] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[24] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, **1**(3):222–238. ACM Press, August 1983.

[25] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, **2**(2):145–154. ACM Press, May 1984.

[26] Adi Shamir. How to share a secret. *Communications of the ACM*, **22**(11):612–613. ACM, November 1979.

[27] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. *Symposium on Reliable Distributed Systems* (10–12 October 1988), pages 93–100. IEEE, 1988.

[28] 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.