Remote Storage with Byzantine Servers

[Extended abstract]

Marcos K. Aguilera
Microsoft Research Silicon Valley
Mountain View, CA, USA

Ram Swaminathan
HP Laboratories
Palo Alto, CA, USA

ABSTRACT

We consider the problem of providing byzantine-tolerant storage in distributed systems where client-server links are much thinner and slower than server-server links. We provide storage algorithms that are unique in two ways. First, our algorithms take into consideration the asymmetry in network connectivity by minimizing client-server communication. To provide this property, we rely on a small amount of partial (eventual) synchrony. Second, our algorithms provide a new property called *limited effect*, which is important for storage systems. To provide the latter property, we use synchronized clocks, which are increasingly common due to GPS devices and NTP, even in otherwise "asynchronous systems" like the Internet. We present two algorithms called QUAD and LINEAR, which provide a trade-off between failure resiliency and efficiency. Our algorithms implement an abortable register [3], which is an abstraction used in some real storage systems, but abortable registers are weaker than atomic registers. Thus, one might wonder if we could have implemented atomic registers instead. We answer this question in the negative: we prove that there are no implementations of atomic registers that provide the limited effect property in systems with failures, even with synchronized clocks.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Design, Reliability, Security, Theory

Keywords

Distributed system, distributed storage, byzantine failures, algorithms, digital signatures

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

1. INTRODUCTION

We consider the problem of providing storage to a distributed system with *byzantine* servers, that is, servers that misbehave due to failures or attacks. The system comprises clients and servers, all connected together by a network. We particularly focus on systems where servers are *remote* with respect to clients, meaning that client-server links have much higher latencies and lower bandwidth than server-server links. For example, clients can be connected to servers via wireless networks, wide area networks, or other thin links, while servers can be connected to each other by expensive network switches, backplanes, or fiber-optical networks. These are common in practice.

We propose new storage algorithms that are unique in two ways. First, they take into consideration the asymmetry in network connectivity by minimizing client-server communication. The client should send just one message to one server and then wait for a reply (in "good" runs with no failures, which are the most common). We do this using a simple common technique: a client signs the request and sends it to a server that acts as a proxy or *coordinator*; the coordinator then communicates with other servers. Because coordinators can be byzantine, they must prove to other servers that they are executing the protocol correctly. This is done with signatures. For this technique to work, we rely on a small amount of partial synchrony so that a client does not wait forever for a dead coordinator.

The second unique aspect of our algorithms is that they provide a property called *limited effect*. This property is motivated by a problem that we have identified with existing distributed storage algorithms that provide linearizability. We call this problem the *destructive pending write* problem. With linearizability, if a client issues a write request and then crashes, the write can take place at an arbitrary time in the future. This allows an adversary to mount an attack to destroy data at any chosen point in the future, say, years after the crash has occurred. All previous linearizable storage algorithms that we know of are vulnerable to this attack. Thus, in addition to linearizability, we require the limited effect property. Roughly speaking, this property prohibits the write of a crashed process from taking effect after the last step of the process. As a result, the effect of writes is limited: they cannot remain pending forever when a client crashes while writing.

Our algorithms ensure the limited effect property by using synchronized clocks—which are increasingly common due to GPS and NTP—and the following technique. Majority-replication algorithms typically rely on a write-back technique on reads: to read a value, a coordinator queries a majority of servers, picks the value with highest timestamp, and writes back this value to a majority of bricks (using the value's original timestamp). This technique ensures that the read value will be seen by subsequent reads. Our

algorithms also rely on the write-back technique, but rather than writing back with a value's original timestamp, they write back with a fresh timestamp. Doing so prevents old pending writes from having the highest timestamp so that they will not take effect in the future.

Writing back with a fresh timestamp, however, creates a vulnerability: a byzantine coordinator can write back very old values with fresh timestamps, and thereby obliterate recently written values. We address this problem by requiring that a coordinator prove to servers that it is writing back an acceptable value. The proof consists of a statement signed by a large number of servers ($n-f$ servers, where n is the number of servers and f is the maximum number that can be byzantine). If a server stores such a proof, it can subsequently prove to clients that the value it has is legitimate. This leads to a new and simple storage algorithm, called *QUAD*, that can tolerate up to $f < n/3$ byzantine servers. However, these proofs are somewhat large vectors with $O(n)$ signatures, which take up space at servers and needs to be verified for each server. As a result, with QUAD, each read or write operation checks $O(n^2)$ signatures, which is expensive and undesirable. We present a second algorithm, called *LINEAR*, that significantly reduce signature usage: servers store only $O(1)$ signatures and each operation checks only $O(n)$ signatures. LINEAR can tolerate up to $f < n/4$ byzantine servers, which is slightly fewer failures than what QUAD tolerates. Thus, QUAD and LINEAR provide a nice tradeoff between resource usage and failure resiliency.

We allow clients to fail by crashing, and servers to fail by becoming byzantine. For example, clients could be closed devices or private machines behind a firewall; servers could be shared machines on the Internet, or general-purpose machines owned by a third-party. We decided against allowing byzantine clients because they can launch an attack in which data is constantly overwritten with garbage, by spuriously sending messages that imitate the write protocol. For instance, even if a single client is byzantine, a read by a correct client can return random data. Thus, our system has a non-byzantine component, the *client*, which is responsible for generating requests. However, we minimize client participation in our algorithms: a client just timestamps, signs and sends its requests to one server, waits for a reply, checks that the reply is valid, and, if not, resends the request to a different server.

Our algorithms implement an abortable register [3], which is a data structure that supports a read and write operation that may sometimes abort their execution in the presence of concurrency[1]. Abortable registers are powerful: they can be used to implement any abortable object [3]. However, abortable registers are weaker than atomic registers, whose operations never abort. Thus, one might wonder if we could have implemented atomic registers instead. We answer this question in the negative: we show that there are no implementations of atomic registers that provide the limited effect property in systems with failures, even if failures are non-byzantine.

Thus, there is a fundamental trade-off between (a) *abortable* registers *with* the limited effect property, and (b) *atomic* registers *without* the limited effect property. With (a), read and write operations may abort if there is contending accesses, but the limited effect property prevents malicious adversaries from overwriting the register via pending writes of crashed processes. With (b), read and write operations never abort, but a malicious adversary can overwrite the contents of the register using pending writes. This situation begs the question: which is better? For storage systems, we think (a) is better for many reasons. First, having aborted operations results in only loss of liveness, whereas overwritten data is loss of safety for the application. Second, if an operation aborts, the operation can simply be retried after a while. Using exponential-backoff, contention eventually stops and the operation no longer aborts. On the other hand, if storage is overwritten, the old contents are lost forever. And third, in a study of several real-world I/O traces of storage systems, [13] found no concurrent conflicting operations to the same block (register). In other words, contention is extremely rare in real-world storage systems, meaning that aborts are very unlikely to occur. This has led to the development of real storage systems that employ abortable registers (e.g., the systems described in [13, 22, 23]) rather than atomic registers.

[1]This is different from a register that provides obstruction freedom [16]: with the latter, concurrent operations may never terminate, whereas with an abortable register, operations always terminate (possibly by aborting).

Algorithm	Low client-server comm.	Limited effect	Resiliency	Register implemented	Uses sync clocks
SBQ–L [20]*	no	no	$f<n/3$	non-abortable	no
Phalanx [19]	no	no	$f<n/4$	non-abortable	no
GWGR [14]	no	no	$f<n/4$	non-abortable	no
CT [7]	no	no	$f<n/3$	non-abortable	no
CL [8]	no	no	$f<n/3$	non-abortable	no
Q/U [1]	no	no	$f<n/5$	non-abortable	no
HQ [10]	no	no	$f<n/3$	non-abortable	no
AAB [4]	no	no	$f<n/3$	non-abortable	no
Zyzzyzva [18]	no	no	$f<n/3$	non-abortable	no
QUAD	yes	yes	$f<n/3$	abortable	yes
LINEAR	yes	yes	$f<n/4$	abortable	yes

*SBQ-L refers to the algorithm in [20] for unreliable asynchronous networks with byzantine clients and self-verifying atomic conformable writes.

Figure 1: Comparison of known algorithms for byzantine-tolerant storage.

In summary, in this paper we study byzantine-tolerant storage systems. We identify the destructive pending write problem, which is present in previously proposed storage algorithms. To address this problem, we introduce the limited effect property and give algorithms that achieve this property. Our algorithms minimize client-server communication, which is important because client-server links tend to be thinner and slower than server-server links. We present two algorithms: the QUAD algorithm checks $O(n^2)$ signatures per operation and tolerates $f < n/3$ byzantine failures; the LINEAR algorithm checks $O(n)$ signatures per operation, and tolerates $f < n/4$ byzantine failures. Both algorithms implement an abortable register. We show that aborting is necessary: there are no implementations of (non-abortable) atomic registers that provide the limited effect property in systems with failures. Because of space limitations, most proofs are omitted from this extended abstract.

2. RELATED WORK

The first practical protocol to implement a byzantine-tolerant state machine is in [8]. Using a state machine, it is easy to implement a storage service. *Q/U* [1] and *HQ* are other protocols for state machines [10], which use a lightweight quorum protocol to improve performance in uncontended cases. Even more efficient state machine protocols have since been proposed, such as Zyzzyzva [18]. *Phalanx* [19] is a distributed storage system that tolerates byzantine servers and clients. In Phalanx, clients broadcast read and write requests to servers, and servers do not communicate with each other. Phalanx also has a version of the protocol for honest clients, which is more efficient. The scheme in [20] also

tolerates byzantine servers and client, where clients broadcast read and write requests to servers, and servers use a mechanism like reliable broadcast to propagate values among themselves. The scheme in [14] uses erasure codes to tolerate failures, which provides better space efficiency than replication. The scheme in [7] uses erasure codes, reliable broadcast and threshold signature schemes. The scheme in [4] uses secret sharing and tolerates byzantine failures of readers and servers. Unlike our algorithms, the above schemes neither guarantee the limited effect property, nor provide low client-server communication (see Figure 1).

It is possible to reduce the client-server communication of the schemes in [7, 8, 10, 19, 20]. For example, with [7, 10, 19, 20], we can (a) run their client protocol at a coordinating server, and (b) have real clients sign their request and forward them to the coordinating server. With [8], a server can collect responses from other servers and then forward them in a single message to a client. These modifications, however, do not provide the limited effect property. For example, with [8], if a client issues a write request and crashes, this request might propagate among a minority M of servers. After many reads that do not involve M, some read that does involve M will cause the write to finally take effect, which violates the limited effect property. In fact, we show later that it is impossible to get an atomic register with the limited effect property, even if processes have synchronized clocks (to timestamp requests). Therefore no variants of existing atomic register algorithms will provide the limited effect property.

There are byzantine-tolerant algorithms for registers writable by just one client (e.g., [2, 6]), but this is a somewhat different service from what we provide. Abortable registers are defined in [3]. Other types of registers that may return an "abort" indication are Δ-registers [11] and ranked registers [9], which are intended to abstract mechanisms for fixing a value in consensus algorithms.

3. INFORMAL MODEL

Our message-passing distributed system has a finite set $\Pi = \Pi^s \cup \Pi^c$ of processes with $\Pi^s \cap \Pi^c = \emptyset$. Processes in Π^s are called *servers*, and processes in Π^c are called *clients*. There are at least two clients and two servers in the system and n is the number of servers. Processes communicate by sending messages over links. There is a link between every pair of servers and between every client and server. There may not be a link between clients, and clients may not be aware of each other.

The system is asynchronous with clocks and a liveness oracle: process speeds and message delays are arbitrary, but clients have synchronized clocks with range $\mathcal{T} = \mathbb{N}$, which need not be related to real time. (The liveness oracle is described below.) In practice, these clocks are reasonable because of GPS devices or NTP, which provides accurate clocks but not predictable network delays[2]. Links are *reliable* (they do not create, drop, or duplicate messages), *encrypted* (only the link endpoints can see data sent through it), and *authenticated* (the receiver knows who sent the message).

Each process (client or server) is an infinite automaton, whose execution proceed in steps. Each step has two actions: (1) receive a message, or send a message, or receive an external input, or issue an external output, or do nothing, and (2) change state. Receiving external inputs and issuing external outputs are actions done only by client processes, to nondeterministically receive an operation request from the environment or output an operation's response, respectively. When a client receives an external input, it becomes *active*. The client remains active until it issues an external output, at which point the client becomes *inactive*. While active, a client does not receive (another) external input. While inactive, a client takes no steps except for a step that receives an external input.

Clients may fail by crashing permanently. We model a crash via a special crash step. A *correct client* is one that does not take a crash step. It takes infinitely many steps or it is active only finitely often (because it wants to execute only finitely many operations). When a client takes a non-crash step, it does so according to its automaton. Servers may fail by becoming byzantine. A *correct server* takes infinitely many steps, and does so according to its automaton. A *faulty client or server* is one that is not correct. Our algorithms require an upper bound f on the number of faulty servers (but not clients): $f < n/3$ for QUAD or $f < n/4$ for LINEAR.

When a client executes an operation, we want it to communicate with just one server in common "good" runs with no failures (we precisely define good runs below). But if this server is unresponsive, the client needs a timeout mechanism so it can try a different server. Thus, we require some amount of partial synchrony, which is abstracted as a *correct server oracle*. A client p accesses the oracle by reading a variable *current-server*$_p$, which has a server id. This variable is changed by the oracle only. The oracle ensures that, for every correct client p, there is a server s that is indistinguishable by p from a correct server, and a time after which *current-server*$_p = s$. Different clients could have different servers. This oracle must be implemented outside our model, using partial synchrony and a simple increasing timeout mechanism.

A *good run* is a run where all processes are correct and, for every client c, *current-server*$_c$ never changes. Intuitively, the latter condition says that the correct server oracle outputs stable information.

Each server p has a pair (e_p, d_p) of public and private keys. All clients have the same pair (e_{client}, d_{client}) of public and private keys. All processes have all public keys. We assume byzantine processes cannot break public-key cryptography.

An implementation $\mathcal{I}$ is a set of automata, one for each process. We omit references to $\mathcal{I}$ when it is clear from context. A *run (of $\mathcal{I}$)* is an infinite sequence of process steps. A *run prefix P (of $\mathcal{I}$)* is a finite sequence of process steps. A *continuation C of P (of $\mathcal{I}$)* is a run that has P as a prefix. Runs, run prefixes, and continuations are subject to obvious well-formedness rules (e.g., steps of processes follow their automata).

4. PROBLEM

We abstract a storage system through *register* objects. An *register* supports two operations, *read*() and *write*(v), such that *read* returns the last value v used in a *write*. An *abortable register* [3] is a variant of a register intended for systems with low contention. With an abortable register, a *read* or *write* operation may abort if it is executed concurrently with another operation (and only in this case). When an operation aborts, it returns a special value $\perp$. The client may later retry the operation if it wants, hoping that the contention would have ceased. If a *write*(v) operation aborts, it may leave the register unchanged or it may change the register's value to v, and which of the two possibilities actually occurs is not indicated to the client. The client can solve this uncertainly by reading the register or writing to the register again. Abortable registers are weaker than the standard registers, but it has been shown to be a powerful abstraction in systems with low contention [3]. It has also been shown that a typical storage system has low contention [13].

We consider wait-freedom as the liveness requirement for all implementations in this paper. The safety requirement is given by linearizability [17]. Roughly speaking, linearizability requires op-

[2]NTP works over "asynchronous networks" like the Internet because NTP only requires small windows of stable network delays to calibrate the local hardware clocks of machines. The success of NTP in the Internet is documented in Chapter 6 of [21].

erations, such as reads or writes, to appear to take effect instantaneously at some point during the *operation interval*: the time from when the operation is invoked until it returns a response. With linearizability, if a client process invokes a write and crashes, this write is *pending*: it may or may not take effect (i.e., change the register's state), and if it takes effect, it can do so at an arbitrary point after the operation's invocation. If a storage implementation is not designed carefully, it could provide a byzantine adversary with the choice of when to effect the pending write. This would allow the byzantine adversary to overwrite useful data at any one time in the future (possibly many days afterward the write was issued), so the storage system would partially lose its ability to retain data safely. This attack is possible with all previous byzantine-tolerant storage algorithms that we know of, including [1, 4, 7, 8, 10, 14, 18–20]. In these protocols, the problem is caused when a write for v starts, a minority M of servers store v, the coordinator crashes, and then there are many reads that do not involve M and hence do not return v, but at some far time in the future, a read that involves M then returns v. This problem cannot be solved with simple techniques like expiration of write requests or expiring tokens.[3]

To prevent infinitely pending writes in storage systems, we require that implementations of registers satisfy the following *limited effect* property. Roughly speaking, if a (client) process starts a write and crashes then the write must appear to take effect before the crash, or never take effect. More precisely, we consider histories with operation requests and responses (as in linearizability), augmented with crash events. Limited effect requires that, for any history H, if a (client) process requests an operation op_1 and it crashes before a (client) process requests an operation op_2 then op_1 is not ordered after op_2 in the linearization of H.

We are interested in (abortable) register implementations that consume low client-server bandwidth. We say that an (abortable) register implementation has *low client-server communication* if, in good runs, a client sends and receives one message per read or write operation. Our goal is to implement abortable registers that (a) are linearizable, (b) have the limited effect property, and (c) have low client-server communication.

5. IMPOSSIBILITY RESULT

We now show that it is impossible to implement an atomic register that guarantees the limited effect property. Intuitively, this impossibility arises because limited effect requires reads to obliterate writes that are pending due to a crash, but such writes are indistinguishable from slow writes.

Theorem 1 *In a system with at least two clients and two servers where a client and a server may crash, there is no implementation of an atomic register satisfying the limited effect property, even if processes have synchronized clocks.*

Proof. Consider a system with at least two clients and two servers where a client and a server may crash. For a contradiction, suppose that there is an implementation $\mathcal{I}$ of an atomic register initialized to 0 that satisfies the limited effect property. The proof outline is as follows. We first define valency for solo continuations in which a client invokes *read*() [12, 15]. We then consider a run R of *write*(1), which is initially 0-valent. We show how to extend R step-by-step, such that the run remains 0-valent forever, and therefore the *write*(1) never terminates—which contradicts the correctness of implementation $\mathcal{I}$.[4]

[3]Although these techniques avoid the specific scenario above, they have bad side effects: they may undo writes that actually completed before expiration, violating linearizability.

[4]The run R we construct will be a run of the model, whether the model has synchronized clocks or not.

Definition 2 *Given a run prefix P, a read-continuation of P is any continuation C of P in which (1) first, an inactive client c invokes read(), (2) then, client c and $n-1$ or n servers take steps until the client receives some response v for its read. We say that v is the value read in C and c is the reader in C.*

Henceforth in this proof, we only consider run prefixes with 0 or 1 active clients (runs prefixes with more active clients may not have continuations since $\mathcal{I}$ only tolerates one client crash). We now use read-continuations to define v-valent prefixes.

Definition 3 *A run prefix P is v-valent if, for every read-continuation C of P, v is the value read in C. A run prefix P is ambivalent if it is not v-valent for any v.*

For example, the empty prefix is 0-valent.

Lemma 4 *Let P be a run prefix where some client c_0 is inactive and let s be a step of some client $c \neq c_0$ applicable to P. If P is v-valent then $P \cdot s$ is not v'-valent for any $v' \neq v$.*

PROOF. Let P be a run prefix where some client c_0 is inactive and let s be a step of some client $c \neq c_0$ applicable to P. By way of contradiction, suppose that P is v-valent and for some $v' \neq v$, $P \cdot s$ is v'-valent. In step s, process c sends 0 or 1 messages and changes its state. If c sends a message, let q be the recipient server, otherwise let q be any server. Since c_0 is inactive in P, there are read-continuations of P in which c_0 is the reader. Consider a read-continuation C of P in which c_0 is the reader and q does not take any steps. Since P is v-valent, v is the value read in C. Because neither q nor c take any steps in C, C is also a read-continuation of $P \cdot s$, and it is one in which v is the value read. Therefore $P \cdot s$ is not v'-valent—a contradiction. □

We now use Lemma 4 to construct a run R in which some client w invokes *write*(1) but it never completes. Intuitively, to construct R, we start with an empty run prefix P_0, which is 0-valent. We let w take a step and apply Lemma 4 to know that the prefix is not 1-valent, and therefore w has not completed its write. Because the prefix is not 1-valent, we find a read-continuation that reads 0. We append this read-continuation to our run; the resulting prefix is 0-valent by the limited effect property. We then append another read-continuation in which all servers take steps, to ensure that, in our final run, all servers take steps forever. We now have a run prefix that is 0-valent but contains one more step of w than P_0. By repeating the construction, we get a run in which w takes infinitely many steps without completing its write.

We now describe R more precisely. Henceforth, let w be some arbitrary but fixed client. We construct a sequence $\{P_i\}$ of run prefixes inductively. In each P_i, there is one write, *write*(1) by client w. P_0 is the run prefix with a single step, in which w receives an external input to *write*(1).

Lemma 5 *P_0 is 0-valent.*

PROOF. In the step of w in P_0, w receives an external input and changes state. Thus, read-continuations of P_0 are indistinguishable from read-continuations where no write occurs. So, P_0 is 0-valent. □

Now assume we have a 0-valent run prefix P_i. Let s_i be some step of w applicable to P_i and Q_i=$P_i \cdot s_i$.

Lemma 6 *Q_i is not 1-valent and w is active in Q_i.*

PROOF. By Lemma 4, $P_i \cdot s_i$ is not 1-valent. Therefore, w has not completed its *write*(1) in $P_i \cdot s_i$ (otherwise subsequent reads

would return 1 and $P_i \cdot s_i$ would be 1-valent). Therefore w is active in $P_i \cdot s_i$. □

Corollary 7 *There is a continuation from Q_i where w takes no steps, and some client c executes read() and returns 0.*

PROOF. Since Q_i is not 1-valent there is a read-continuation of Q_i whose value read is $v \neq 1$ and whose reader is not w. Then v must be 0 because 1 is the only value that has been written in Q_i. □

Definition 8 *Let S be a continuation from Q_i from Corollary 7 where some client c_i executes read() and it returns 0. Let T_i be the run prefix of S up to when c_i finishes its read.*

Lemma 9 *T_i is 0-valent.*

PROOF. T_i has only one write, namely, *write*(1) by client w. Note that the last step of w in T_i is before c_i starts *read*(). By the limited effect property, if w takes no further steps then its *write* cannot take effect after its last step. Therefore, any reads that start from T_i must return 0, so T_i is 0-valent. □

In the *read*() just added to T_i, one server might not take any steps. We now append to T_i a *read* where all servers take steps. The resulting run prefix will still be 0-valent since T_i is 0-valent.

Definition 10 *Let S be a continuation from T_i where some client c executes read() and it returns 0. Let P_{i+1} be the run prefix of S up to when c finishes its read. (Note that P_{i+1} is a continuation of T_i.)*

Lemma 11 *P_{i+1} is 0-valent.*

PROOF. By Lemma 9, T_i is 0-valent. Thus, P_{i+1} is also 0-valent. □

This finishes our inductive construction: we started with a 0-valent prefix P_i and constructed a 0-valent prefix P_{i+1} where w takes another step without completing its *write*(1). Taking the limit, we get a run R in which w takes infinitely many steps and never completes its *write*(1). Thus, Theorem 1 is proved.

We note that the impossibility result holds for wait-free implementations; it leaves open the possibility of implementations with weaker liveness guarantees, such as obstruction freedom.

6. ALGORITHM THAT TOLERATES CRASH FAILURES

We first explain a simpler algorithm for abortable registers that tolerates crash failures and provides the limited effect property. This algorithm requires that $f < n/2$, i.e., a majority of servers are correct.

Our algorithm is derived from the algorithm by Attiya, Bar-Noy, and Dolev [5], which we call the *ABD-algorithm* and summarize now. Basically, each server stores a $(value, timestamp)$ pair. To write a value v, a client proceeds in two phases. In phase 1, the client asks servers for their stored timestamp, waits for $n-f$ timestamps, and picks a new timestamp T that is larger. In phase 2, the client asks servers to store (v, T). To read a value, a client also proceeds in two phases. In phase 1, the client asks servers for their stored value-timestamp pairs, waits for $n-f$ replies, and picks the pair (v^*, T^*) with highest timestamp. In phase 2, the client writes back (v^*, T^*) to the servers and waits for $n-f$ acknowledgements.

To write a value v:
1. client obtains a unique timestamp T
2. client sends v and T to the current coordinator c=*current-server*$_p$
3. coordinator c sends v and T to all servers s (* **write phase** *)
4. if a server s saw any timestamp $>T$, it returns $\perp$
   (* 'saw' means 'ever received in any messages containing...' *)
5. else s stores (v, T) and returns *ok*
6. c waits for $n - f$ replies
7. if some reply is $\perp$ then c returns $\perp$
8. else c returns *ok*
9. client waits for reply or change of coordinator (*current-server*$_p$)
10. if change of coordinator then goto 2

To read a value:
1. client obtains a unique timestamp T
2. client sends T to the current coordinator c=*current-server*$_p$
3. coordinator c sends T to all servers s (* **read phase** *)
4. if a server s saw timestamp $>T$, it returns $\perp$
5. else s returns its stored value and timestamp
6. c waits for $n - f$ replies
7. if some reply is $\perp$ then c returns $\perp$
8. c picks the reply value v^* with largest timestamp
9. c sends v^* and T to all servers s (* **write phase** *)
10. if s saw timestamp $>T$, it returns $\perp$
11. else s stores (v^*, T) and returns *ok*
12. c waits for $n - f$ replies
13. if some reply is $\perp$ then c returns $\perp$
14. else c returns v^*
15. client waits for reply or change of coordinator (*current-server*$_p$)
16. if change of coordinator then goto 2

Figure 2: Algorithm that tolerates crash failures.

The client then returns v^* as the value read. Phase 2 is needed to ensure that v^* is stored at a majority of servers, in case the write of v^* is in progress or did not finish.

Our algorithm, shown in Figure 2, employs two trivial modifications to the ABD-algorithm: (1) A client does not talk to all servers. Instead, it sends its read or write command to one server, which then acts as a proxy/coordinator for the client. This saves client-server bandwidth. (2) When a client writes, it obtains its new timestamp from its synchronized clock instead of querying the servers. This saves one phase when writing.

We also make a fundamental change to the ABD-algorithm, to obtain the limited effect property: in phase 2 of the read protocol, we pick a new fresh timestamp to write back v^* instead of writing back with the original timestamp of v^* as in the ABD-algorithm. This is called *timestamp promotion* and it ensures the limited effect property: if there is a lurking pending write for value v_{bad} then a subsequent read operation will cause the value read to be written back with a higher timestamp than v_{bad}, making it impossible for v_{bad} to take effect subsequently. However, timestamp promotion breaks linearizability, as the following scenario shows: (a) the register's initial value is 0, (b) a write of 1 starts with timestamp 1, (c) a read also starts, finds 0 at $n-f$ server, and writes back 0 with timestamp 2, and (d) the write of 1 completes. In this case, subsequent reads will return 0 (which now has timestamp 2), even though the write of 1 completed. We solve this problem by causing the write of 1 to return $\perp$ (abort)—this is shown, for example, in line 4 of the algorithm for writing or reading. Here, if a server has seen previously a request with higher timestamp then it returns $\perp$, causing the coordinator to also return $\perp$ (abort). We can do that because we need only implement an abortable register, in which concurrent operations can abort.

7. THE QUAD ALGORITHM

We now extend the algorithm of Section 6 to tolerate byzantine servers (including byzantine coordinators) as long as $f < n/3$. We use a simple and common idea: the client, who is always honest, signs requests that servers can execute with confidence, while servers sign responses so that the client know its request was fulfilled. For example, to write v to the register, a client signs a request with v and a new timestamp T, and sends it to the coordinator. The coordinator then asks servers to store v and T, attaching the client's signature. Servers also store this signature, to be used later in reads. It is important that the signature includes both v and T, not just v, for otherwise a malicious coordinator could overwrite new values with old values. Each server responds with a signed acknowledgement that it stored v and T, which the coordinator returns to the client as proof of execution of the write operation.

The protocol to read the register is more complex because of timestamp promotion: after querying $n-f$ servers and picking the value v^* with largest timestamp, the coordinator needs to write back v^* with the new timestamp T, but there is no client signature authorizing that.[5] We need to prevent the coordinator from cheating on the write-back value. The coordinator could ask the client to sign a request for v^* and T, but this scheme requires extra client-server communication, which we want to avoid. In our algorithm, the coordinator sends to servers the set S of $n-f$ replies from which v^* is picked. Each server can then validate the write-back by looking at S and verifying that the coordinator chose v^* correctly. To prevent the coordinator from forging S, each of the $n-f$ replies in S must be signed by a different server.

This solution, however, has a problem: if a byzantine coordinator happens to receive more than $n-f$ replies, it could generate two different sets S_1 and S_2 of $n-f$ replies each, such that the v^* of each set is different. By doing so, the coordinator can convince different servers to write back different values. We solve this problem by adding a **certification phase** to the protocol, which forces the coordinator to certify a value-timestamp pair before it is stored at server. At most one value-timestamp pair can be certified for a given timestamp. This phase works as follows: (1) the coordinator sends S, v^* and T (the new timestamp) to all servers; (2) Servers check that v^* is computed correctly from S and reply with a signed statement including S, v^* and T. An honest server signs at most one statement for a given T; to keep track of that, a server remembers the largest T used in a statement it previously signed, and it rejects signing statements with smaller or equal timestamps (Timestamps T are signed by clients so that a byzantine server cannot cause servers to reject signing statements for the largest timestamp.) (3) The coordinator collects signed statements from $n-f$ servers into a set *valproof*, called a *certificate*. Intuitively, the certificate serves as proof that v^* can be promoted to timestamp T. Thus, the coordinator attaches the certificate to the write-back request of v^*, and each server then verifies that the certificate is correct (all statements refer to v^* and T, and they are signed by $n-f$ servers). If so, the server stores v^*, T, and the certificate. The server needs to store the certificate so that later, when it replies to a read request, it can prove that its value-timestamp pair (v^*, T) is legitimate. In other words, a server can either store a (v, T) that comes from a write, or a (v, T) that comes from a write-back. In the first case, there is a client signature on (v, T), and in the second case, there is a certificate for (v, T).

The algorithm is given in Fig. 3 and its timeline is shown in Fig. 4. Due to space limitations, the full pseudo-code of the al-

[5]Note that the client previously signed a request to write v^* with timestamp T^* but not with the new timestamp T.

To write a value v:

1. client gets a unique signed timestamp T
2. *sig* := client signs (*WRITE*, v, T)
3. client sends (v, T, *sig*) to the coordinator c=*current-server*$_p$

(* **write phase** *)

4. coordinator c sends (v, T, *sig*) to all servers s
5. a server s checks (v, T, *sig*); if bad then return $\perp$
6. if s saw timestamp$>T$, it returns ($\perp$, *highest seen timestamp*)
7. else s stores (v, T, *sig*) and returns signed (*ok*, v, T)
8. c waits for $n - f$ valid replies
9. if some valid reply is ($\perp, T'$), c returns T'
10. c sets *stoproof* := { $n-f$ valid replies }
11. c returns (*ok*, *stoproof*)
12. client waits for reply or change of coordinator (*current-server*$_p$)
13. if bad reply or coordinator changes then goto 2

To read a value:

1. client gets a unique signed timestamp T
2. *sig* := client signs (*READ*, T)
3. client sends (T, *sig*) to the coordinator c=*current-server*$_p$

(* **read phase** *)

4. coordinator c sends (T, *sig*) to all servers s
5. a server s checks (T, *sig*); if bad then return $\perp$
6. if s saw timestamp$>T$ or s saw T in this phase before then s returns ($\perp$, *highest seen timestamp*)
7. *bindsig* := s signs (*BIND*, T, *timestamp of stored value*)
8. s returns stored value, timestamp, proof, and *bindsig*
9. c waits for $n - f$ valid replies
10. if some valid reply is ($\perp, T'$), c returns ($\perp, T'$)
11. c sets *maxproof* := set of *bindsigs* in the $n-f$ valid replies
12. c picks the valid reply (v^*, T^*, p^*, b^*) with largest T^*

(* **certification phase** *)

13. c sends (v^*, T, T^*, p^*, *maxproof*, *sig*) to all servers s
14. s checks (v^*, T, T^*, p^*, *maxproof*, *sig*); if bad return $\perp$
15. if s saw timestamp$>T$ or s saw T in this phase before then s returns ($\perp$, *highest seen timestamp*)
16. s returns a signature on (*ACKVAL*, v^*, T)
17. c collects $n - f$ valid replies
18. if some valid reply is ($\perp, T'$), c returns ($\perp, T'$)
19. c sets valproof := set of $n - f$ valid replies

(* **write phase** *)

20. c sends (v^*, T, *valproof*) to all servers s
21. s checks (v^*, T, *valproof*); if bad then return $\perp$
22. if s saw timestamp$>T$, it returns ($\perp$, *highest seen timestamp*)
23. else s stores (v^*, T, *valproof*) and returns signed (*ok*, v^*, T)
24. c waits for $n - f$ valid replies
25. if some valid reply is ($\perp, T'$), c returns T'
26. c sets *stoproof* to the $n - f$ valid replies
27. c returns (v^*, *stoproof*)
28. client waits for reply or change of coordinator (*current-server*$_p$)
29. if bad reply or coordinator changes then goto 1

Figure 3: QUAD **algorithm for byzantine failures.**

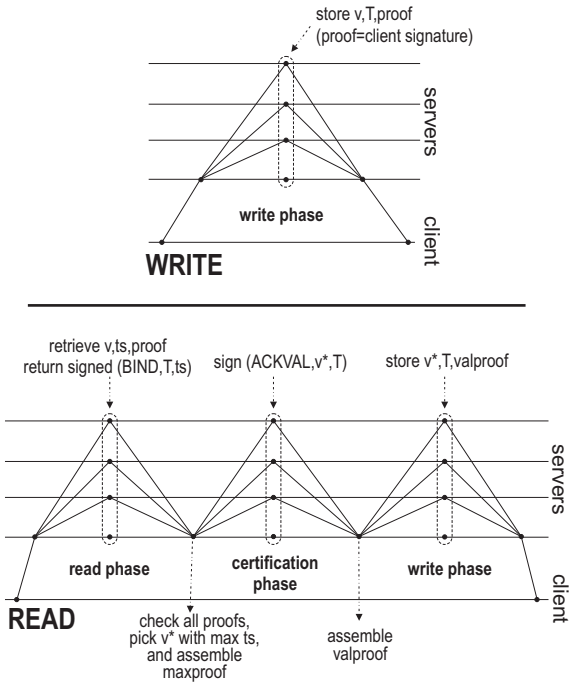

Figure 4: Timeline depiction of QUAD algorithm.

1. Initially, all servers hold a value v with timestamp T.
2. Then, a byzantine server changes its stored value to some old value $\hat{v}$ but with a new timestamp $T_0 > T$. Timestamp T_0 comes from a client that started a read but then crashed.
3. Next, a client requests a write for v_1 with timestamp $T_1 > T_0$, the request goes to a byzantine coordinator, the coordinator only sends (v_1, T_1) to one correct server, and the client crashes,
4. Similarly for values $v_2, \ldots, v_f$: for each value, some client requests a write for v_j with timestamp $T_j > T_{j-1}$, the request goes to a byzantine coordinator, the coordinator only sends (v_j, T_j) to a correct server (and the server is different for each j), and the client crashes.
5. After all this, we have f correct servers holding values $v_1, \ldots, v_f$ with timestamps $T_1, \ldots, T_f$, respectively, and one byzantine server holding value $\hat{v}$ with timestamp T_0. If a read occurs next, winning-rule attempt 1 incorrectly picks $\hat{v}$ as the value to be returned to the client. But the only acceptable values that could be picked (according to linearizability) is v or one of the v_j's.

Figure 5: Scenario where winning-rule attempt 1 breaks.

gorithm (which includes detailed checking of message formatting, and signatures, which are straightforward but extensive) is omitted from this paper.

Theorem 12 *Consider a system with n servers and k clients where up to $f < n/3$ servers can be byzantine and any number of clients can crash. The* QUAD *algorithm implements an abortable register and satisfies the limited effect property. In good runs, for each operation a client sends and receives only one message and servers check $O(n^2)$ signatures.*

8. THE LINEAR ALGORITHM

With the QUAD algorithm, space at each server is $\Theta(n)$ (since a server may have to store a certificate with a signature from $n-f$ servers) and reading a value may involve checking all the signatures of $n-f$ certificates, for a total of $\Theta(n^2)$ signatures. In the LINEAR algorithm, we reduce signature usage significantly: servers do not store certificates and reading does not require checking certificates. As a result, space at each server is $O(1)$ and operations check $O(n)$ signatures. There is a trade-off: the algorithm requires tolerates up to $f < n/4$ failures instead of $f < n/3$.

To understand how the LINEAR algorithm works, consider what might happen to the QUAD algorithm if servers did not keep certificates and read coordinators did not check them. Then, a byzantine server could falsely claim that an old value has been promoted to a new timestamp. If this were to happen, the next read request would return an old value (which was promoted to the highest timestamp), and this would violate linearizability. It appears that this problem is solved by requiring timestamps to be signed by clients, but this does not help: a client may sign a new timestamp for reading, send this timestamp to a byzantine coordinator, and then crash. Now a byzantine server has a signed timestamp, and so the attack mentioned above is possible.

The QUAD algorithm solved this problem by using certificates to prevent byzantine servers from promoting old values to new timestamps. The LINEAR algorithm uses a different solution: it employs a new mechanism to pick the "winning value" in a read operation, instead of picking the value with highest timestamp. The mechanism ensures that even if byzantine servers promote old values to new timestamps, these values are not picked by a read coordinator, even if the coordinator cannot tell that these values were maliciously promoted.

To understand how this mechanism works, first note that there are at most f byzantine servers. Therefore, the read coordinator might use the following rule to choose the value that will be returned:

Winning-rule attempt 1: *Order values by timestamp breaking ties arbitrarily, discard the top f values, pick the top value that is left.*

The intuition is that, after a value is written or written-back, it is stored with the highest timestamp at $n-f$ servers. Later, if f byzantine servers try to promote old values to larger timestamps, the $(f+1)$-th top value is still an honest value. This mechanism, however, breaks under a slightly more sophisticated attack, shown in Fig. 5.

Another natural rule might be the following:

Winning-rule attempt 2: *Discard values stored at less than $f+1$ servers; among the values left, pick the one with highest timestamp.*

The intuition is that, since there are only f byzantine servers, the above rule discards any maliciously-promoted values that those f servers might hold. The problem here, however, is that this rule might end up discarding all values: there is a scenario in which each server (including correct ones) ends up with a different value. This scenario uses the idea in steps (3) and (4) of Fig. 5: a client starts a write, sends its request to byzantine coordinator, which stores the

value at a single server, and then the client crashes.

It is unclear that there exists a correct winning rule that uses only the current timestamp of a value. In our LINEAR algorithm, the winning rule uses two timestamps: the current one, and the one used originally to write the value. For example, suppose v is first written with timestamp T_1 and, later, a write-back promotes v's timestamp to T_2. Then, each server stores v, T_1 and T_2, where T_1 is called the *left timestamp* of v, and T_2 is the *right timestamp* of v. If T_1 has not been promoted, then $T_2 = T_1$. Note that servers do *not* keep the entire history of timestamps of a value: they keep only the original timestamp and the latest promoted timestamp. For example, if a subsequent write-back promotes v's timestamp to T_3, then only T_1 and T_3 are stored, not T_2. A left timestamp comes from a write operation, and there is a client signature that binds the timestamp to the value being written. A right timestamp, if different from the left timestamp, comes from the timestamp promotion in a read operation; there is client signature on the timestamp, but it does not bind it to any values. We often combine the left and right timestamps into a pair $[T_1, T_2]$ or into a triple $[T_1, T_2, v]$, where v is the value bound to T_1.

The LINEAR algorithm uses the following rule (below we give some intuition of why it works):

Winning rule: *(1) Among the $n-f$ triples obtained from servers, find the set, called candSet, of $2f+1$ triples with largest right timestamp, breaking ties arbitrarily. (2) If some timestamp T_0 is the left timestamp of $f+1$ or more triples in candSet, pick any such triple as winner. (3) Otherwise, pick the triple in candSet with largest left timestamp, breaking ties arbitrarily.*

The algorithm is shown in Figure 6. Due to space limitations, the algorithm's full pseudo-code (which includes detailed checking of message formatting, and signatures, which are straightforward but extensive) is omitted from this paper. The algorithm ensures the following key property:

Theorem 13 *In any run, if some read or write operation succeeds, resulting in $n-2f$ correct servers storing the same triple $[T_1, T_2, v]$, then afterwards the winning rule will never select an old, stale value (one whose left timestamp is less than T_1).*

Thus, a read always return a relatively recent value, and this implies linearizability. We now provide a proof sketch of why the above property holds.

The following is an important property of triples stored at servers:

Lemma 14 *Suppose some set S_1 of $n-2f$ correct servers store the same triple $[T_1, T_2, v]$. If some correct server ever stores a triple $[T_1', T_2', v']$ with $T_2' > T_2$ then either $T_1' = T_1$ or $T_1' > T_2$.*

With this lemma, we show Theorem 13 as follows. Suppose some set S_1 of $n-2f$ correct servers store the triple $[T_1, T_2, v]$. Later, suppose we apply the winning rule for a set S_2 of $n-f$ triples (each triple from a different server), and consider the *candSet* computed in the rule. Then (1) *candSet* has at least one triple from a server in S_1 since *candSet* has $2f+1$ elements, and (2) S_2 has at least $n-3f$ elements from S_1. Since $f < n/4$, we have $n-3f \geq f+1$. There are two cases:

Case 1. Assume that some timestamp T_0 is the left timestamp of $f+1$ or more triples in *candSet*—as in part (2) of the winning rule. From (2), we have that (3) S_2 has at least $f+1$ elements from S_1, which are all correct servers. Let *goodCandSet* be the triples in *candSet* from correct servers. Since *candSet* has $2f+1$ triples, *goodCandSet* has at least $f+1$ triples. Servers in S_1 cannot replace their right timestamps with a timestamp smaller than T_2, since a correct server rejects requests to store right timestamps lower than

To write a value v:

∗ 1. client gets unique signed timestamp T tagged with 'W'
(* tag ensures timestamp is used only as left timestamp *)
2. *sig* := client signs (*WRITE*, v, T)
3. client sends (v, T, T, *sig*) to the coordinator c=*current-server*$_p$
(* **write phase** *)
∗ 4. coordinator c sends (v, T, T, *sig*) to all servers s
∗ 5. a server s checks (v, T, T, *sig*); if bad then return $\perp$
6. if s saw timestamp$>T$, it returns ($\perp$, *highest seen timestamp*)
∗ 7. else s stores (v, T, T, *sig*) into (*Value*, *Left-ts*, *Right-ts*, *Sig*) and returns signed (*ok*, v, T)
8. c waits for $n-f$ valid replies
9. if some valid reply is ($\perp$, T'), c returns T'
10. c sets *stoproof* := { $n-f$ valid replies }
11. c returns (*ok*, *stoproof*)
12. client waits for reply or change of coordinator (*current-server*$_p$)
13. if bad reply or coordinator changes then goto 2

To read a value:

∗ 1. client gets unique signed timestamp T tagged with 'R'
2. *sig* := client signs (*READ*, T)
3. client sends (T, *sig*) to the coordinator c=*current-server*$_p$
(* **read phase** *)
4. coordinator c sends (T, *sig*) to all servers s
5. a server s checks (T, *sig*); if bad then return $\perp$
6. if s saw timestamp$>T$ or s saw T in this phase before then s returns ($\perp$, *highest seen timestamp*)
∗ 7. *bindsig* := s signs (*BIND*, T, *Left-ts*, *Right-ts*)
∗ 8. s returns (*Value*, *Left-ts*, *Right-ts*, *Sig*, *bindsig*)
9. c waits for $n-f$ valid replies
10. if some valid reply is ($\perp$, T'), c returns ($\perp$, T')
11. c sets *reps* := set of $n-f$ valid replies
∗ 12. c uses winning rule to pick v^*, *left-ts**, *right-ts** and associated *sig**, *bindsig**
(* **certification phase** *)
∗ 13. c sends (v^*, T, *sig**, *reps*, *sig*) to all servers s
∗ 14. s uses winning rule to recompute v^* from *reps* and checks if (v^*, T, *sig**, *reps*, *sig*) is valid; if bad return $\perp$
15. if s saw timestamp$>T$ or s saw T in this phase before then s returns ($\perp$, *highest seen timestamp*)
16. s returns a signature on (*ACKVAL*, v^*, T)
17. c collects $n-f$ valid replies
18. if some valid reply is ($\perp$, T'), c returns ($\perp$, T')
19. c sets valproof := set of $n-f$ valid replies
(* **write phase** *)
∗ 20. c sends (v^*, *left-ts**, T, *sig**, *valproof*) to all servers s
∗ 21. s checks (v^*, *left-ts**, T, *sig**, *valproof*); if bad, return $\perp$
22. if s saw timestamp$>T$, it returns ($\perp$, *highest seen timestamp*)
∗ 23. else s stores (v^*, *left-ts**, T, *sig**) into (*Value*, *Left-ts*, *Right-ts*, *Sig*) and returns signed (*ok*, v, T)
24. c waits for $n-f$ valid replies
25. if some valid reply is ($\perp$, T'), c returns T'
26. c sets *stoproof* to the $n-f$ valid replies
27. c returns (v^*, *stoproof*)
28. client waits for reply or change of coordinator (*current-server*$_p$)
29. if bad reply or coordinator changes then goto 1

Figure 6: LINEAR **algorithm that tolerates byzantine failures. Asterisks indicate changes relative to the** QUAD **algorithm.**

its own. Thus, from (3), *goodCandSet* has at least $f+1$ triples with right timestamps equal to T_2 or greater. If such a triple has right timestamp greater than T_2 then, by Lemma 14, its left timestamps is either T_1 or greater than T_2. If such a triple has right timestamp equal to T_2 then its left timestamp is equal to T_1 (since (a) when a read coordinator is upgrading timestamps to T_2, it must commit to a single value and such a value is v, and (b) the left timestamp of a triple is bound to its value through a client signature). Note that there are at most f triples in *candSet* that are not in *goodCandSet*. Therefore, timestamp T_0 (the timestamp which is the left timestamp of $f+1$ or more triples in *candSet*) is either equal to T_1 or it is greater than T_2. Thus, the winning rule does not choose a triple whose left timestamp is less than T_1.

Case 2. Now assume that no such timestamp T_0 exists, i.e., part (3) of the winning rule applies. By (1), *candSet* has at least one triple from a server in S_1. Let p be such a server. If p changes its triple from $[T_1, T_2, v]$ to something else, then its right timestamp increases, so by Lemma 14, its left timestamp either remains as T_1 or increases beyond T_2. Therefore, the largest left timestamp in triples in *candSet* is at least T_1. Thus, the winning rule does not choose a triple whose left timestamp is less than T_1.

This shows Theorem 13. It is worth noting that Theorem 13 does not hold if we change the winning rule so that *candSet* had $2f+2$ instead of $2f+1$ triples with largest timestamp. Intuitively, the reason is that part (2) of the rule could be triggered for a timestamp T_0 smaller than T_1 in the argument above.

Theorem 15 *Consider a system with n servers and k clients where up to $f < n/4$ servers can be byzantine and any number of clients can crash. The* LINEAR *algorithm implements an abortable register and satisfies the limited effect property. In good runs, for each operation a client sends and receives only one message and servers check $O(n)$ signatures.*

9. CONCLUSION

We considered the problem of handling byzantine servers in distributed storage systems. We presented algorithms for abortable registers that ensure the limited effect property, while minimizing the communication between clients and servers. These algorithms trade off resiliency for efficiency: the first algorithm tolerates $f<n/3$ failures and checks $O(n^2)$ signatures per operation, while the second algorithm tolerates $f<n/4$ failures and checks $O(n)$ signatures per operation. Some interesting questions for future work are the following. Is there an algorithm providing the best resiliency and efficiency of both our algorithms? Is it possible to avoid the use of synchronized clocks without adding phases of communication? Is it possible to do reads in fewer than three phases? Our algorithms ensure that operations always terminate (wait-freedom); are there weaker liveness conditions (e.g., obstruction-freedom) that allow providing the limited effect property for an atomic register?

Acknowledgements

We thank the anonymous reviewers for many helpful comments.

10. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Symposium on Operating Systems Principles*, pages 59–74, Oct. 2005.

[2] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Wait-free regular storage from byzantine components. *Information Processing Letters*, 101(2):60–65, Jan. 2007.

[3] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *Symposium on Principles of Distributed Computing*, pages 23–32, Aug. 2007.

[4] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *International Symposium on Distributed Computing*, pages 7–19, Sept. 2007.

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

[6] H. Attiya and A. Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. *Parallel Processing Letters*, 16(4):419–428, Dec. 2006.

[7] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *International Conference on Dependable Systems and Networks*, pages 115–124, June 2006.

[8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, pages 173–186, Feb. 1999.

[9] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *ACM Symposium on Principles of Distributed Computing*, pages 78–87, July 2002.

[10] J. Cowling, D. Meyers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Symposium on operating systems design and implementation*, pages 177–190, Dec. 2006. Longer version available as MIT Technical Report MIT-CSAIL-TR-2007-009.

[11] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *International Symposium on Distributed Computing*, pages 133–147, Oct. 2002.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[13] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *International Conference on Dependable Systems and Networks*, pages 125–134, June 2004.

[14] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems*, pages 135–144, June 2004.

[15] D. Hendler and N. Shavit. Operation-valency and the cost of coordination. In *Symposium on Principles of Distributed Computing*, pages 84–91, July 2003.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, May 2003.

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

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong.

Zyzzyva: speculative byzantine fault tolerance. In *Symposium on Operating Systems Principles*, pages 45–58, Oct. 2007.

[19] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *IEEE Symposium on Reliable Distributed Systems*, pages 51–60, Oct. 1998.

[20] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–326, Oct. 2002.

[21] D. L. Mills. *Computer Network Time Synchronization: the Network Time Protocol*. CRC Press, 2006.

[22] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building reliable enterprise storage systems on a shoestring. In *Workshop on Hot Topics in Operating Systems*, pages 169–174, May 2003.

[23] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *International conference on Architectural support for programming languages and operating systems*, pages 48–58, Oct. 2004.