Byzantine Clients Rendered Harmless

Barbara Liskov and Rodrigo Rodrigues
MIT CSAIL and INESC-ID
32 Vassar Street - Cambridge, MA 02139 - USA
Rua Alves Redol 9 - 1000 Lisboa - Portugal

Abstract

Byzantine quorum systems have been proposed that work properly even when up to f replicas fail arbitrarily. However, these systems are not so successful when confronted with Byzantine faulty clients. This paper presents novel protocols that provide atomic semantics despite Byzantine clients. Our protocols are the first to handle all problems caused by Byzantine clients. They prevent Byzantine clients from interfering with good clients: bad clients cannot prevent good clients from completing reads and writes, and they cannot cause good clients to see inconsistencies. In addition we also prevent bad clients that have been removed from operation from leaving behind more than a bounded number of writes that could be done on their behalf by a colluder.

Our protocols are designed to work in an asynchronous system like the Internet and they are highly efficient. We require $3f + 1$ replicas, and either two or three phases to do writes; reads normally complete in one phase and require no more than two phases, no matter what the bad clients are doing.

We also present strong correctness conditions for systems with Byzantine clients that limit what can be done on behalf of bad clients once they leave the system. Furthermore we prove that our protocols are both safe (they meet those conditions) and live.

1 Introduction

Quorum systems [4, 14] are valuable tools for building highly available replicated data services. A quorum system can be defined as a set of sets (called quorums) with certain intersection properties. These systems allow read and write operations to be performed only at a quorum of the servers, since the intersection properties ensure that any read operation will have access to the most recent value that was written.

The original work on quorum systems assumed that servers fail benignly, i.e., by crashing or omitting some steps. More recently, researchers have developed techniques that enable quorum systems to provide data availability even in the presence of arbitrary (Byzantine) faults [9]. Earlier work provides correct semantics despite server (i.e., replica) failures and also handles some of the problems of Byzantine clients [9, 10, 5, 1, 2, 11, 12].

This paper extends this earlier work in two important ways. First, it defines new protocols that handle all problems of Byzantine clients. In addition, our protocols are more efficient than previous proposals, in terms of both operation latency and number of replicas. Second, the paper states correctness conditions for such systems and proves that protocols are correct. The correctness conditions are stronger than what has been stated previously [11] and what has been guaranteed by previous approaches.

Since a dishonest client can write garbage into the shared variable, it may seem there is little value in limiting what bad clients can do. But this is not the case, for two reasons. First, bad clients can cause a protocol to misbehave so that good clients are unable to perform operations (i.e., the protocol is no longer live) or observe incorrect behavior. For example, if the variable is write-once, a good client might observe that its state changes multiple times.

Second, bad clients can continue to interfere with good ones even after they have been removed from operation, e.g., by a system administrator who learns of the misbehavior. We would like to limit such interference so that, after only a limited number of writes by good clients, any *lurking writes* left behind by a bad client will no longer be visible to good clients. A lurking write is a modification launched by the bad client before it was removed from operation that will happen (possibly with help from an accomplice) after it has left the system. By limiting such writes we can ensure that the object becomes useful again after the departure of the bad client, e.g., some invariant that good clients preserve will hold.

Of course, it is not possible to prevent actions by a bad client, even if it has been shut down, if the private key it uses to prove that it is authorized to modify an object can be used by other nodes; thus, a bad client is in the system as long as any node knows its private key. If nodes can expose their private keys, we can make statements about behavior in the absence of a bad client only when all faulty nodes have been removed from operation [11] – a condition that is not very useful in practice. A more practical approach is make use of secure cryptographic coprocessors (like the IBM 4758 [7]) that allow signing without exposing the private key. Our correctness condition is stated in a way that allows either model about the lifetime of faulty nodes in the system.

Thus, we make the following contributions:

- We present strong correctness conditions for an atomic read/write object in a system in which both clients and replicas can be Byzantine. Our conditions ensure atomicity for good clients, and also limit the effects of bad clients that have been removed from operation: our conditions bound the number of lurking writes by a constant factor, and prevent lurking writes after bad clients stop and good clients subsequently overwrite the data.

- We present the first Byzantine quorum protocols that satisfy the conditions using only $3f+1$ replicas (to survive f faulty replicas) and work in an unreliable asynchronous network like the Internet. Furthermore our protocols are efficient: To do writes requires either 3 phases (our *base protocol*) or mostly 2 phases (our *optimized protocol*). Reads usually require 1 phase; they sometimes need an additional phase (to write back the data just read). Our base protocol ensures that there can be at most one lurking write after a bad client has left the system; the optimized protocol ensures slightly weaker behavior: there can be a most two lurking writes.

- We are able to manage with a small number of replicas and phases because our protocol makes use of "proofs", an approach that we believe can be used to advantage in other protocols. A proof is a collection of $2f+1$ certificates from different replicas that vouch for some fact, e.g., that a client has completed its previous write, or that the state read from a single replica is valid.

- We prove the correctness of our protocols, both safety and liveness. In addition we describe a variation of our protocols that supports a still stronger correctness condition, in which we can bound the number of writes of good clients that it takes to ensure that modifications of the bad client that has left the system will no longer be visible. This variation sometimes requires an additional phase to do a write.

The rest of this paper is organized as follows. We being by describing our assumptions about the system. Section 3 describes our base protocol. Section 4 describes our correctness conditions and we prove our base protocol meets them in Section 5. Section 6 describes our optimized protocol and proves its correctness. Section 7 describes a variation of our protocols that allows us to bound the number of overwrites needed to hide effects of lurking writes. Section 8 discusses related work and we conclude in Section 9.

2 Model

The system consists of a set $\mathcal{C} = \{c_1, ..., c_n\}$ of client processes and a set $\mathcal{S} = \{s_1, ..., s_n\}$ of server processes. Client and server processes are classified as either correct or faulty. Correct processes are constrained to obey their specification, i.e., they follow the prescribed algorithms. Faulty processes may deviate arbitrarily from their specification, i.e., we assume a Byzantine failure model [8]. Note that faulty processes include those that fail benignly, and those that are initially correct and fail at some point in the execution of the system.

We refer to the set of faulty clients as $\mathcal{C}_{bad}$ and the set of correct clients as $\mathcal{C}_{ok}$, and consequently we have $\mathcal{C} = \mathcal{C}_{bad} \cup \mathcal{C}_{ok}$ (and respectively $\mathcal{S} = \mathcal{S}_{bad} \cup \mathcal{S}_{ok}$). Note that our algorithms do not require the knowledge of $\mathcal{C}_{ok}$, $\mathcal{C}_{bad}$, $\mathcal{S}_{ok}$, nor $\mathcal{S}_{bad}$. In other words, we do not assume that we can detect faults.

We assume an asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, corrupt them, or deliver them out of order, and there are no known bounds on message delays or on the time to execute operations. We assume the network is fully connected, i.e., given a node identifier, any other node can (attempt to) contact the first node directly by sending it a message.

For liveness, we require that if a message is retransmitted repeatedly it will eventually be delivered. Note that we only require this for liveness; safety does not depend on any assumptions a about message delivery.

We assume nodes can use unforgeable digital signatures to authenticate communication. More precisely, any node, n, can authenticate messages it sends by signing them. We denote a message m signed by n as $\langle m \rangle_{\sigma_n}$. And (with high probability) no node can send $\langle m \rangle_{\sigma_n}$ (either directly or as part of another message) on the network for any value of m, unless it is repeating a message that has been sent before or it knows $n's$ private key. We call the set of possible signatures Σ. These signatures can be verified with public keys in a set P.

We also assume the existence of a collision-resistant hash function, h, such that any node can compute a digest $h(m)$ of a message m and (with high probability) it is impossible to find two distinct messages m and m' such that $h(m) = h(m')$.

These assumptions are probabilistic but there exist signature schemes (e.g., RSA [13]) and hash functions (e.g., SHA-1 [3]) for which they are believed to hold with very high probability. Therefore, we will assume they hold with probability one in the rest of the paper.

To avoid replay attacks we tag certain messages with nonces that are signed in the replies. We also assume that when clients pick nonces they will not choose a repeated nonce with probability one.

3 BFT-BC Algorithm

This section presents our construction for a read/write variable implemented using Byzantine quorum replication, and that tolerates Byzantine-faulty clients. We begin by giving a brief overview of Byzantine quorums in Section 3.1. Then we present our *base protocol*. This protocol requires 3 phases to write; in Section 6 we present an optimization that requires only 2 phases most of the time.

3.1 Byzantine Quorums Overview

This section gives an overview of how current algorithms use Byzantine quorums to implement a shared read/write variable. This presentation follows the original BQS protocol [9], using their construction for a system that doesn't handle Byzantine clients. (We discuss this system further in Section 8.)

A Byzantine quorum system defines a set of subsets of a replica group with certain intersection properties. A typical way to configure such a system is to use groups of $3f + 1$ replicas to survive f failures with quorums of size $2f + 1$ replicas. This ensures that any two quorums intersect in at least one non-faulty replica. Each of the replicas maintains a copy of the data object, along with an associated timestamp, and a client signature that authenticates the data and timestamp.

Two phases are required to *write* the data. First, the client contacts a quorum to obtain the highest timestamp produced so far. The client then picks a timestamp higher than what was returned in the first phase, signs the new value and timestamp, and proceeds to the second phase where the new value is stored at a quorum of replicas.

Replicas allow write requests only from authorized clients. A replica overwrites what it has stored only if the timestamp in the request is greater than what it already has.

The *read* protocol usually has a single phase where the client queries a quorum of replicas and returns the value with the highest timestamp (provided the signature is correct). An extension of this protocol [10] requires a second phase that writes back the highest value read to a quorum of replicas (this ensures atomic semantics for reads).

3.2 BFT-BC Protocol

The protocol just presented is not designed to handle Byzantine-faulty clients, which can cause damage to the system in several ways, e.g.:

1. Not follow the protocol by writing different values associated with the same timestamp.
2. Only carry out the protocol partially, e.g., install a modification at just one replica.
3. Choose a very large timestamp and exhaust the timestamp space.
4. Issue a large number of write requests and hand them off to a colluder who will run them after the bad client has been removed from the system. This colluder could be one of the replicas, or a completely separate machine.

To avoid these problems, we need to limit what bad clients can do. But we must also allow good clients to make progress, in spite of bad clients that might collude with one another and/or with bad replicas.

Our protocol accomplishes these goals. It uses $3f+1$ replicas, and quorums can be any subset with $2f+1$ replicas. It uses a three-phase protocol to write, consisting of a read phase to obtain the most recent timestamp, a prepare phase in which a client announces its intention to write a particular timestamp and value, and a write phase in which the client does the write that it prepared previously.

As it moves from one phase to another, however, the client needs to "prove" that what it is doing is legitimate. A proof takes the form of a quorum of certificates from different replicas that vouch for some fact.

For example, the purpose of the prepare phase is for the client to inform the replicas of what it intends to do, and this must be acceptable, e.g., the timestamp it proposes can't be too big. Replicas that approve the prepare request return certificates that together provide a "prepare proof". This proof is needed to carry out the write phase: a client can only carry out a write that has been approved. If a write is allowed by a replica it returns a certificate and these certificates together form a "write proof". This proof is needed for the replica to do its next write: it cannot do a second write without completing its first one. This constraint, plus the fact that proofs cannot be forged or predicted in advance by bad clients, is what limits the number of lurking writes a bad client can leave behind when it stops (to be carried out by some node that colludes with it).

We now describe our protocol in detail.

As mentioned, each object in BFT-BC is replicated at a set of $3f+1$ replicas, numbered from 0 to $3f$. Quorums can be any subset with $2f+1$ replicas and we use a three-phase protocol to write.

Each replica maintains the following per-object information:

- *data*, the value of the object.
- *current-ts*, the associated timestamp.
- *prepare-proof*, a valid prepare proof for above timestamp and a hash of the value.
- *prepare-list*, a list containing the timestamp, hash, and proposing client of currently prepared writes.
- *write-ts*, the timestamp of the latest write known to have completed at $2f+1$ replicas.

In the remainder of this paper we simplify the presentation by considering a system containing only a single object, and therefore we omit object identifiers from the description of the protocol.

3.2.1 Write Protocol

Our protocols require that different clients choose different timestamps, and therefore we construct timestamps by concatenating a sequence number with a client identifier: $ts = \langle ts.val, ts.id \rangle$. We assume that client identifiers are unique. To increment a timestamp a client with identifier c uses the following function: $succ(ts, c) = \langle ts.val + 1, c \rangle$.

Timestamps can be compared in the usual way, by comparing the val parts and if these agree, comparing the client ids.

Client Processing

To write a data object, clients go through a three-phase protocol. In all phases, clients retransmit their requests to account for lost messages; they stop retransmitting once they collect a quorum of valid replies.

Phase 1. The client, c, sends a READ-TS request to all replicas; the request contains a nonce to avoid replays. The replies each include a prepare proof for the timestamp being returned. The client waits for a quorum of correctly authenticated replies. It selects the largest timestamp t_{max} it received, computes a new timestamp $t' = succ(t_{max}, c)$, and moves to phase 2.

Phase 2. The client sends PREPARE requests to all replicas. The request contains the new timestamp, the prepare proof obtained in phase 1, a hash of the value it intends to write, and a write proof (for the most recent write done by this client, or null if this is the first write done by this client), all authenticated by the client. The client waits for a quorum of PREPARE-ACKs, and then moves to phase 3; the $2f + 1$ responses it received in phase 2 constitute a prepare proof for the write.

Phase 3. The client sends a WRITE request containing the prepare proof along with the new value, and waits for a quorum of valid replies, all authenticated by the client.

Each reply contains a signed statement vouching for the write for that timestamp and the replica id of the sender. A vector of $2f + 1$ such statements constitutes a write proof for the current operation.

Processing at replicas

Phase 1. Replicas reply to a READ-TS request by sending the prepare proof for their current timestamp and value, plus the nonce, all authenticated by the replica.

Phase 2. Phase 2 processing is the crucial part of the algorithm. The replicas check to ensure that the timestamp being proposed is reasonable, that the client is doing just one prepare, that the value being proposed doesn't differ from a previous request for the same timestamp, and that the client has completed its previous write.

The replica discards the message if it is invalid: if the prepare or write proof can't be validated, the timestamp doesn't have the client id in the low-order part, the proposed timestamp isn't the successor of the timestamp in the prepare proof, or the client isn't authorized to write the object.

Next, the replica uses the write proof to update its *write-ts*: it sets this to the larger of the timestamp in the write proof it just received, and what it already has stored. Then it removes any entry from its *prepare-list* whose timestamp is less than or equal to the *write-ts*.

If there is still an entry in the *prepare-list* for the client, the replica discards the request unless that entry matches what is being proposed (both the timestamp and the hash).

If the replica accepts the request it updates the prepare-list by adding the timestamp and hash for this client (unless this information is already in the list). Then it returns a PREPARE-ACK message containing a signed certificate for the new timestamp and hash.

Phase 3. Replicas carry out the request only if it is valid: the prepare proof must be valid and the new value must match the hash in the proof. Replicas respond to valid requests by sending a WRITE-ACK containing a signed certificate for the new timestamp. The replica additionally modifies its state if the timestamp in the request is greater than its current timestamp; in this case it stores the provided information as the current value, timestamp, and associated prepare proof.

3.2.2 Read Protocol

The read protocol usually requires just one phase.

Phase 1. The client sends a READ request to all replicas; the request contains a nonce. A replica replies with its value, timestamp, prepare proof, and nonce all signed by it. The client waits for a quorum of valid responses and chooses the one with the largest timestamp (this is the return value). If all the timestamps are the same the read protocol ends.

Phase 2. Otherwise the client goes on to the write-back phase for the largest timestamp; this is identical to phase 3 when a client is doing a write, except that the client needs to send only to replicas that are behind, and it must wait only for enough responses to ensure that $2f + 1$ replicas now have the new information.

3.3 Discussion

In the above description we mentioned that certain messages or statements were authenticated, but the kind of authentication that may be used was unspecified. This issue is important since different techniques have different costs: we can authenticate a point-to-point message by means of symmetric cryptography by establishing session keys and using message authentication codes (MACs). This does not work for signing statements that have to be shown to other parties, in which case we need to rely on more expensive public key cryptography.

Our protocol requires signing using public key cryptography in two places: the phase 2 and phase 3 responses. These signatures are needed because they are used as proofs offered to third parties, e.g., the prepare proof is generated for one client but then used by a different client to justify its choice of the next timestamp.

A further point is that only the phase 2 response signature needs to happen in the foreground. The signature for the phase 3 response can be done in the background: a replica can do this after replying to the phase 2 request, so that it will have the signature ready when the phase 3 request arrives.

4 Correctness Condition

This section defines the correctness conditions for a variable shared by multiple clients that may incur Byzantine failures. We begin by defining histories (Section 4.1), and then we give our correctness condition.

4.1 Histories and Stopping

We use a definition of history similar to the one proposed in [6], extended to handle Byzantine clients.

The execution is modeled by a history, which is a sequence of events of the following types:

- Invocation of operations.
- Response to operation invocations.
- Stop events.

An invocation by a client c is written $\langle c : x.op\rangle$ where x is an object name and *op* is an operation name (possibly including arguments). A response to c is written $\langle c : x.rtval\rangle$ where $rtval$ is the response value. A response *matches* an invocation if their object names agree and their client names agree. A stop event by client c is written $\langle c : stop\rangle$.

A history is *sequential* if it begins with an invocation, if every response is immediately followed by an invocation (or a stop or no event), and if every invocation is immediately followed by a matching response. A client subhistory $H|c$ of a history H is the subsequence of all events in H whose client names are c. A history is *well-formed* if for each client c, $H|c$ is sequential. We use $\mathcal{H}$ to denote the set of well-formed histories.

A object subhistory history $H|x$ is the subsequence of all events whose object names are x and a history H is a single-object history for some object x if $H|x = H$. A *sequential specification* for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification of x.

An operation o in a history is a pair consisting of an invocation $inv(o)$ and the next matching response $rsp(o)$. A history H induces an irreflexive partial order $<_H$ on the operations and stop events in H as follows: $o_0 <_H o_1$ if and only if $rsp(o_0)$ precedes $inv(o_1)$ in H; $o_0 <_H \langle c : stop\rangle$ if and only if $rsp(o_0)$ precedes $\langle c : stop\rangle$ in H; $\langle c : stop\rangle <_H o_1$ if and only if $\langle c : stop\rangle$ precedes $inv(o_1)$ in H; and $\langle c_1 : stop\rangle <_H \langle c_2 : stop\rangle$ if and only if $\langle c_1 : stop\rangle$ precedes $\langle c_2 : stop\rangle$ in H.

4.1.1 Verifiable Histories

In [6], the notion of linearizability is applied to a concurrent computation, which is modeled by a history (a finite sequence of invocation and response events by all processes).

It would be difficult to model such computations in our environment, since faulty processes do not obey any specification, and therefore we cannot define what an invocation or a response means for such processes. However, we know that after a STOP event from a faulty process it will halt.

Therefore we introduce the concept of a *verifiable history* which is a history that contains the sequence of invocations and responses from correct clients, and stop events from faulty clients.

The correctness condition we present next is applicable only to verifiable histories.

4.2 Correctness Condition

We are now ready to define the correctness condition for variables shared by multiple processes that may incur Byzantine failures.

The idea is that, as in linearizability [6], we require that the verifiable history look plausible to the correct processes, i.e., that there is a sequential history in which all processes are correct that explains what the correct processes observed. Furthermore, we require that once a faulty client stops, its subsequent effect on the system is limited in the following way: The number of operations by that client that are "seen" by correct clients after it stopped is bounded by some constant.

These concepts can be formalized as follows.

Definition 1 A verifiable history $H \in \mathcal{H}$ is BFT-linearizable if there exists some legal sequential history $H' \in \mathcal{H}$ such that

1. $H|p = H'|p, \forall p \in \mathcal{C}_{ok}$
2. $<_H \subseteq <_{H'}$
3. $\forall c \in \mathcal{C}_{bad} : \langle c : stop \rangle \in H \Rightarrow$
   $(\exists h_1, h_2 \subseteq H' : H' = h_1 \langle c : stop \rangle h_2 \wedge$
   $|\{o \in h_2 : o = \langle c : x.op \rangle\}| \leq max\text{-}b)$

Points 1 and 2 of the definition above state that there must exist a sequential history that looks the same to correct processes as the verifiable history in question, and that sequential history must preserve the $<_H$ ordering (in other words, if an operation or a stop event precedes another operation or stop event the verifiable history, then the precedence must also hold in the sequential history).

Point 3 says that if all faulty clients stop, then when we look at the sequential history and consider the subhistory after the last stop event by faulty clients (this is h_2), the number of events by faulty clients in that history is bounded by *max-b*.

Note that there is a counter-intuitive notion that a bad client can execute operations after it has stopped. This corresponds to the notion that the bad client left some pending operations (e.g., with a colluder) before it left, and this is precisely the effect we want to minimize.

We can now define a BFT-linearizable object to be an object whose verifiable histories are BFT-linearizable with respect to its sequential specification.

5 Correctness Proof

This section sketches a proof that the algorithm presented in Section 3 meets the correctness conditions presented in Section 4.

The idea is to look at what kind of properties we can ensure given a certain state of the system when a faulty client c stops. If we look at the *current-ts* values stored at that instant, we can guarantee the following.

Lemma 1. Let ts_{max} be the $f+1$st highest timestamp stored by non-faulty replicas at time t_{stop} (when some faulty client c stops). Then the following must hold

1. At any time up to t_{stop}, no node can collect a write proof for a timestamp t' such that $t' > ts_{max}$.
2. There are no two timestamps $t_1, t_2 > ts_{max}$ such that client c assembled a prepare proof for t_1 and t_2.
3. No two prepare proofs exist for the same timestamp $t > ts_{max}$ and different associated values.

Proof.

1. By algorithm construction, a nonfaulty replica will not sign an entry in a write proof vouching for a timestamp higher than the one held in the variable *current-ts*. Since non-faulty replicas always store increasing timestamp values, this means that the number of signatures that can be held in the system at time t_{stop} for timestamps higher than ts_{max} is at most $2f$ (i.e., f from faulty replicas and the f correct replicas that may hold timestamps higher than ts_{max}).
2. By contradiction, suppose that client c held prepare proofs for t_1, t_2, both greater than ts_{max}. The two proofs intersect in at least one nonfaulty replica. By part (1) of this lemma, that replica had its *write-ts* variable always at a value less than or equal to ts_{max} (at all times up to and including t_{stop}). Therefore that replica could not have signed both proofs, since after signing the first one (say, for t_1) it would insert an entry for client c and t_1 in its prepare-list, and that entry would not be removed (because of the value of *write-ts*), which prevents the replica from signing the proof for t_2.
3. Suppose, by contradiction that two prepare proofs exist for timestamp t and values v and v'. By the quorum intersection properties, these two prepare proofs contain at least one signature from the same correct replica. By part (1) of this lemma, no write proof was ever assembled for timestamps greater or equal than ts_{max}, and these entries in the prepare list were never removed. This violates the constraint that correct replicas do not sign a timestamp that is already in its prepare list for a different value.

$\square$

We are now ready to show the correctness of our algorithm.

Theorem 1. The BFT-BC algorithm is BFT-linearizable.

Proof. Consider any correct reader in the case that the writer is correct. In this case, the quorum initially accessed in a read operation intersects the quorum written in the most recently completed write operation in at least one correct replica. Therefore, the read returns either the value in the most recently completed write, or a value with a higher timestamp (which could be written concurrently with the read). Since a read also writes back its obtained value and timestamp to a quorum of processes, any subsequent read by a correct reader will return that timestamp value or a later one. So, for any execution, we construct the history needed to show BFT-linearizability by putting every read right after the write whose value it returns.

If the writer is faulty, we construct the sequential history to show BFT-linearizability as follows: for each read by a correct reader returning v such that the phase 3 request for v was produced by client c_b, insert a write operation in the history that writes v (by client c_b) immediately before the read.

Insert a stop event before the invocation of the first operation that succeeded the stop event in the original (verifiable) history (i.e., as late as possible while preserving the $<_H$ dependencies).

It is left to show that if a faulty process stops, this history contains a bounded number of operations by that faulty process after it stops.

To prove this condition we note that Lemma 1 part (2) says that client c only assembled prepare proofs for a single timestamp for a write that would become visible after it stopped (i.e., with a timestamp greater than ts_{max}), and Lemma 1 part (3) implies that if the write were to become visible, the prepare proof could only correspond to a single value.

This means the number of operations by the faulty client c after it stops is at most one.

5.1 Liveness

As far as liveness is concerned, we guarantee that good clients can always execute read operations in the time it takes for two client RPCs to complete at $2f+1$ replicas (i.e., the requests reaching the replicas and the respective replies returning to the client). This is so because at least $2f+1$ replicas will provide them with the appropriate answers (client requests are answered unconditionally provided the requests are well-formed). For the write protocol, the operation will complete in the time for three client RPCs to complete in $2f+1$ replicas, and the trick is to ensure that their phase 2 requests are never refused (since phase 1 requests are answered unconditionally and phase 3 requests are answered if they contain a valid prepare proof, which good clients will always send).

The phase 2 request also get replies since the client will submit the write proof for its latest write alongside with it, which will force the replica to discard the client's entry in the prepare list and accept the prepare request.

6 Optimized BFT-BC Algorithm

This section describes our optimized protocol, which usually requires only two phases to do a write, and proves that the optimized protocol satisfies our correctness condition.

6.1 Optimized BFT-BC

Our plan is to avoid one of the phases by merging it with another phase. We cannot merge phases 2 and 3 since the protocol requires that the prepare information be stored at $f+1$ honest replicas before the client can do phase 3: this is how we avoid a bad client being able to create many writes in advance that could be launched after it leaves the system. Therefore, we will merge phases 1 and 2.

The idea is that the client sends the hash in the phase 1 request and the replica does phase 2 on its behalf: it predicts the next timestamp, adds the new timestamp with the hash to the prepare-list, and returns this information, signed by itself. If the client receives a quorum of responses all for a particular (highest) new timestamp, it can move to phase 3 immediately.

This optimization will work well in the normal case where a write is actually received by all replicas. Therefore the good client is highly likely to receive a quorum of acknowledgments for the same timestamp in the responses in phase 1, and most of the time a write will require two phases.

However there are problems that must be overcome for the optimization to be used. They occur when there are concurrent writes or when other clients are faulty.

A simple example is the following. Suppose a client does phase 1 for some hash h. It receives PREPARE-ACKS for a particular timestamp t from replicas $R1$ and $R2$, and PREPARE-ACKS for a larger timestamp t' from replicas $R3$ and $R4$. The client will then be unable to get approval for either t or t', because to approve the request a replica would need to violate our restriction on having at most one entry per client on the *prepare-list*.

Clearly we need to find a way to allow clients to do writes in cases like this one. But given our current constraints this is not possible.

To allow the client to make progress we will weaken our constraint on how many entries a client can have in the prepare list. In particular we will change the replica state to have a second prepare list, *opt-list*, containing entries produced by the replica, with the constraint that a client can have at most one entry per list. The two entries for the same client (one in each list) might be for different timestamps, or they might be for the same timestamp. In the latter case, it will be possible for the write with that timestamp to happen twice (when there is a bad client). In this case we need a way to order the writes: we will do this by using the numeric order on their hashes.

6.2 Detailed Write Protocol

Phase 1. The client, c, sends a READ-TS-PREP request containing the hash of the proposed value, and the client's current write proof, to all replicas.

Replicas process the request in the usual phase-2 way, including removing entries from the prepare-lists. Then it will do the prepare on behalf of the client for timestamp $t' = succ(ts, c)$, unless the client already has an entry in either prepare list for a different timestamp or hash. If the prepare is successful, the replica adds an entry to the *opt-list* for t' and the hash (unless that entry is already in the list) and returns a PREPARE-ACK signed by it; otherwise it returns a normal phase 1 response.

If the client gets a quorum of PREPARE-ACKS (i.e., obtains a prepare proof) for the same new timestamp, it moves to phase 3.

Phase 2. Otherwise the client chooses the new timestamp as before and carries out phase 2 for it (as in the normal protocol). When it has a quorum of signatures for this choice (obtained either in phase 1 or phase 2), it moves to phase 3.

Replicas handle phase 2 requests as in the normal protocol; they ignore the *opt-list* in this processing.

Phase 3. The client does phase 3 in the normal way.

The replica also does normal processing, except that the timestamp in the write might match the current timestamp, but with a different value: in this case it retains the value with the larger hash.

6.3 Discussion

The optimized protocol can lead to a client having entries on two prepare lists (normal and optimized). A dishonest client can exploit this to carry out phase 3 twice as part of the same write request. And as a result, each dishonest client can leave two lurking writes behind when it is removed from the system.

Another point is that it is now possible for honest clients to see valid responses to a read request that have the same timestamp but different values. The client protocol resolves this situation by returning (and writing back) the value with the larger hash.

6.4 Optimized BFT-BC Correctness Proof

For the optimized protocol some of the invariants shown in Section 5 need to be slightly modified. Notably, Lemma 1 parts (2) and (3) no longer hold, since a faulty client can now collect two distinct prepare proofs (for its entries in *prepare-list* and *opt-list*). Therefore Lemma 1 parts (2) and (3) become:

Lemma 1'.
$(2+3)$. For the optimized BFT-BC protocol, no more than two prepare proofs can exist for distinct timestamp, value pairs, with timestamps greater than ts_{max}.

Proof. Same as Lemma 1, but taking into consideration that the new algorithm allows for one entry in the normal prepare list, and another in the optimistic prepare list.

□

This only affects the proof of Theorem 1 in that the number of operations by a faulty client after it stops becomes 2 instead of 1, and therefore the main theorem still holds:

Theorem 2. The optimized BFT-BC algorithm is BFT-linearizable.

7 Stronger Correctness Conditions

In this section we propose a stronger correctness condition than the one introduced in Section 4, and discuss how to extend the current protocol to meet the new condition.

7.1 New Correctness Conditions

The idea is that we want to further limit the effects a bad client can have on the system once it stopped such that, if correct clients execute a certain number of write operations after a faulty client stops, then no more operations by that client will ever be "seen". (We can generalize this to any state-overwriting operation in a set $O_{overwrite}$ in case the variable has an interface other than read and write.)

We formalize the BFT-linearizable+ condition as being equal to the condition presented in Section 4, except that point (3) is modified to state that no operations by the faulty client c appear after the kth state-overwriting operation in h_2.

7.2 Modified BFT-BC protocol

To meet this stronger correctness condition, we merely need to modify the BFT-BC protocol to require the client to submit a write proof in its prepare request (along with the information it already sends in the prepare). This proof certifies that the timestamp it is proposing is the successor of one that corresponds to a write that has already happened; a replica will discard the request if this condition doesn't hold.

The client can easily assemble this proof in phase 1 if all responses to its phase 1 request contain the same timestamp. If they don't the client can obtain the proof by doing a write-back; the difficulty is getting the value to write back. This could be accomplished by having phase 1 actually be a read, but this is unattractive since values can be large. So instead, the client fetches the value, if necessary, in a separate step; effectively it redoes phase 1 as a normal read, although this can be optimized (e.g., to fetch from the $f + 1$ replicas that returned the largest timestamps).

This scheme guarantees that the timestamp in the lurking write is the successor of a lower bound on the value stored by at least $f + 1$ non-faulty replicas when the bad client stopped. Consequently, if there were two consecutive writes by correct clients after the bad client stopped, the lurking write would no longer be seen.

8 Related Work

In this section we discuss the previous work that dealt with bad clients, and also the previous work on correctness conditions.

Our protocol is more efficient than those proposed previously. Furthermore it enforces stronger constraints on the behavior of bad clients: we handle all problems handled by previous protocols plus we limit the number of lurking writes that a bad client can leave behind after it has left the system.

In addition we provide stronger safety and liveness guarantees than previous protocols: in particular read operations cannot return null values, and reads terminate in a constant number of rounds, independently of the behavior of concurrent writers.

Our use of prepare proofs is what enables us to ensure liveness, since this allows a client to complete a read using information it received from just one replica. No previous protocol uses this technique. As a result they can only ensure that a read will return the result of the latest write in a constant number of rounds provided there are no concurrent writes. Yet concurrent writes are a real issue in a system with Byzantine clients, since bad clients can use this to prevent good clients from making progress: they can either write continually, or they can write partially (not complete their writes), and either of these causes problems for readers.

The previous proposals fall into two categories: those that ensure that at most one value can be associated with a timestamp, and those that enforce non-skipping timestamps.

The initial work on Byzantine quorum systems [9] described the quorum consensus protocol discussed in Section 3, which used $3f+1$ replicas, one phase reads and two phase writes, and did not handle Byzantine clients. That paper also described a protocol for a system of $4f + 1$ replicas that prevented malicious clients from associating different values with different timestamps; the protocol required a three-phase communication among the replicas to be carried out as part of performing a write (in addition to the client-server communication) where servers guarantee that each value and timestamp pair is propagated to a quorum. The protocol provides liveness, but at the expense of providing weak semantics for reads where they could return a null value in case of concurrent writes.

The Phalanx system [10] improves on the previous result in two ways. First it added the write-back phase (to the simpler protocol) as a way to ensure atomicity for reads. In addition it presented a more efficient protocol to handle Byzantine clients. That protocol used $4f + 1$ replicas, but clients carried out a three-phase protocol to do a write, and the server-to-server communication was no longer needed. In the new protocol for Byzantine clients, read operations could return a null value if there was an incomplete or a concurrent write.

The work by Goodson et al. [5] proposes a solution for erasure-coded storage that tolerates Byzantine failures of clients and servers; this system requires $4f + 1$ replicas. This work is focused on integrating erasure coded storage with Byzantine quorum protocols and they tolerate Byzantine clients that write different "values" to different replicas by having the next reader detect the inconsistency and, if possible, repair it. In some cases, reads may return a null value.

The work of Martin et al. [12] proposes a protocol that only uses $3f + 1$ replicas (like our protocol). They require a quorum of $2f + 1$ identical replies for read operations to succeed, which is difficult to ensure in an asynchronous system. Their solution is to assume a reliable asynchronous network model, where each message is delivered to all correct replicas. This means that infinite retransmission buffers are needed in an asynchronous environment like the Internet: the failure of a single replica (which might just have crashed) causes all messages from that point on to be remembered and retransmitted. In this protocol concurrent writers can slow down readers. The authors discuss how to extend their protocols to guarantee that Byzantine clients associate the same value with the same timestamp by having servers forward the writes among themselves (again, this is possible due to their network model) and keep the highest value for each timestamp. They also discuss how to prevent a faulty client from exhausting resources at correct servers, but at the expense of possibly sacrificing liveness for correct readers.

Two papers (Bazzi and Ding [1], Cachin and Tessaro [2]) describe protocols that enforce non-skipping timestamps.

In both cases, the protocols are based on the Martin et al. work, which uses the reliable asynchronous network model with the problems we mentioned above. The protocol of Bazzi and Ding [1] requires $4f+1$ replicas. Cachin and Tessaro [2] require $3f+1$ replicas; an added problem in this system is the use of threshold signatures to vouch for an increment on the current timestamp (since these signatures are very expensive). They also employ an expensive reliable broadcast mechanism for client-server communication.

Our definition of BFT-linearizability presented in Section 4 builds upon a condition (called Byznearizability) proposed by Malkhi et al. [11]. This work was the first to point out the problem of lurking writes. However, their correctness condition is weaker than ours, since they require only that the number of lurking writes is finite, whereas we require that the number of lurking writes is bounded by a constant. In fact, their correctness condition is met by the variant of the Phalanx protocol [10] that was designed to handle only honest clients. Another point is that this paper did not consider the use of secure co-processors to prevent malicious clients from leaking their private key, and therefore their condition requires all faulty clients and servers to stop in order to provide any guarantees.

Our condition is also stronger due to the fact that we consider the number of overwrites to mask all the lurking writes (see Section 7).

9 Conclusions

This paper has presented novel protocols for Byzantine quorum systems that provide atomic semantics despite Byzantine faulty clients. Ours is the first protocol to fully address the complete range of problems that can arise with Byzantine clients. We prevent bad clients from leaving behind more than one or two lurking writes and from exhausting the timestamp space. In addition Byzantine clients are unable to interfere with good clients in the sense that they cannot prevent good clients from completing reads and writes, and they cannot cause good clients to see inconsistencies. Another point is that an extension of our protocol can additionally ensure that the effects of a Byzantine client are no longer visible to good clients at all after two successive writes by good clients (or four successive writes in the optimized protocol).

We also presented strong correctness conditions that address the above problems: we require that protocols guarantee atomicity for good clients, and limit what can be done on behalf of bad clients once they leave the system. Furthermore we proved that our protocols are both safe (they meet those conditions) and live.

Our protocols are designed to work in an asynchronous system like the Internet and they are highly efficient. Our base protocol completes writes in three network round-trips; the optimized protocol reduces this cost so that writes normally complete in two network round-trips (at the expense of allowing one more lurking write). In either case reads normally complete in one phase, and require no more than two phases, no matter what the bad clients are doing. We achieve these efficiencies because of our use of "proofs", which allow clients or replicas to know that information presented to them is valid, without having to hear the same thing directly from $2f+1$ replicas.

References

[1] R. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *Distributed Computing, 18th International Conference, DISC 2004*, pages 405–419, Oct. 2004.

[2] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. Technical Report RZ 3575, IBM Research, Feb. 2005.

[3] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[4] D. K. Gifford. Weighted voting for replicated data. In *17th SOSP*.

[5] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2004.

[6] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, Munich, Germany, Jan. 1987.

[7] IBM. http://www.ibm.com/security/cryptocards/, 2005.

[8] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[9] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[10] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. 17th SRDS*, Oct. 1998.

[11] D. Malkhi, M. Reiter, and N. Lynch. A Correctness Condition for Memory Shared by Byzantine Processes. Unpublished manuscript, Sept. 1998.

[12] J. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. Technical Report TR-02-38, University of Texas at Austin, Department of Computer Sciences, Aug. 2002.

[13] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[14] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.