Appears in the *Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999*

Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu

Abstract

This paper describes a new replication algorithm that is able to tolerate Byzantine faults. We believe that Byzantine-fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior. Whereas previous algorithms assumed a synchronous system or were too slow to be used in practice, the algorithm described in this paper is practical: it works in asynchronous environments like the Internet and incorporates several important optimizations that improve the response time of previous algorithms by more than an order of magnitude. We implemented a Byzantine-fault-tolerant NFS service using our algorithm and measured its performance. The results show that our service is only 3% slower than a standard unreplicated NFS.

1 Introduction

Malicious attacks and software errors are increasingly common. The growing reliance of industry and government on online information services makes malicious attacks more attractive and makes the consequences of successful attacks more serious. In addition, the number of software errors is increasing due to the growth in size and complexity of software. Since malicious attacks and software errors can cause faulty nodes to exhibit Byzantine (i.e., arbitrary) behavior, Byzantine-fault-tolerant algorithms are increasingly important.

This paper presents a new, *practical* algorithm for state machine replication [17, 34] that tolerates Byzantine faults. The algorithm offers both liveness and safety provided at most $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas are simultaneously faulty. This means that clients eventually receive replies to their requests and those replies are correct according to linearizability [14, 4]. The algorithm works in asynchronous systems like the Internet and it incorporates important optimizations that enable it to perform efficiently.

There is a significant body of work on agreement and replication techniques that tolerate Byzantine faults (starting with [19]). However, most earlier work (e.g., [3, 24, 10]) either concerns techniques designed to demonstrate theoretical feasibility that are too inefficient to be used in practice, or assumes synchrony, i.e., relies on known bounds on message delays and process speeds. The systems closest to ours, Rampart [30] and SecureRing [16], were designed to be practical, but they rely on the synchrony assumption for correctness, which is dangerous in the presence of malicious attacks. An attacker may compromise the safety of a service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the replica group. Such a denial-of-service attack is generally easier than gaining control over a non-faulty node.

Our algorithm is not vulnerable to this type of attack because it does not rely on synchrony for safety. In addition, it improves the performance of Rampart and SecureRing by more than an order of magnitude as explained in Section 7. It uses only one message round trip to execute read-only operations and two to execute read-write operations. Also, it uses an efficient authentication scheme based on message authentication codes during normal operation; public-key cryptography, which was cited as the major latency [29] and throughput [22] bottleneck in Rampart, is used only when there are faults.

To evaluate our approach, we implemented a replication library and used it to implement a real service: a Byzantine-fault-tolerant distributed file system that supports the NFS protocol. We used the Andrew benchmark [15] to evaluate the performance of our system. The results show that our system is only 3% slower than the standard NFS daemon in the Digital Unix kernel during normal-case operation.

Thus, the paper makes the following contributions:

- It describes the first state-machine replication protocol that correctly survives Byzantine faults in asynchronous networks.
- It describes a number of important optimizations that allow the algorithm to perform well so that it can be used in real systems.

This research was supported in part by DARPA under contract DABT63-95-C-005, monitored by Army Fort Huachuca, and under contract F30602-98-1-0237, monitored by the Air Force Research Laboratory, and in part by NEC. Miguel Castro was partially supported by a PRAXIS XXI fellowship.

- It describes the implementation of a Byzantine-fault-tolerant distributed file system.
- It provides experimental results that quantify the cost of the replication technique.

The remainder of the paper is organized as follows. We begin by describing our system model, including our failure assumptions. Section 3 describes the problem solved by the algorithm and states correctness conditions. The algorithm is described in Section 4 and some important optimizations are described in Section 5. Section 6 describes our replication library and how we used it to implement a Byzantine-fault-tolerant NFS. Section 7 presents the results of our experiments. Section 8 discusses related work. We conclude with a summary of what we have accomplished and a discussion of future research directions.

2 System Model

We assume an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the restriction mentioned below. We assume independent node failures. For this assumption to be true in the presence of malicious attacks, some steps need to be taken, e.g., each node should run different implementations of the service code and operating system and should have a different root password and a different administrator. It is possible to obtain different implementations from the same code base [28] and for low degrees of replication one can buy operating systems from different vendors. N-version programming, i.e., different teams of programmers produce different implementations, is another option for some services.

We use cryptographic techniques to prevent spoofing and replays and to detect corrupted messages. Our messages contain public-key signatures [33], message authentication codes [36], and message digests produced by collision-resistant hash functions [32]. We denote a message m signed by node i as $\langle m \rangle_{\sigma_i}$ and the digest of message m by $D(m)$. We follow the common practice of signing a digest of a message and appending it to the plaintext of the message rather than signing the full message ($\langle m \rangle_{\sigma_i}$ should be interpreted in this way). All replicas know the others' public keys to verify signatures.

We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely. We also assume that the adversary (and the faulty nodes it controls) are computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques mentioned above. For example, the adversary cannot produce a valid signature of a non-faulty node, compute the information summarized by a digest from the digest, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties [33, 36, 32].

3 Service Properties

Our algorithm can be used to implement any deterministic replicated *service* with a *state* and some *operations*. The operations are not restricted to simple reads or writes of portions of the service state; they can perform arbitrary deterministic computations using the state and operation arguments. Clients issue requests to the replicated service to invoke operations and block waiting for a reply. The replicated service is implemented by n replicas. Clients and replicas are non-faulty if they follow the algorithm in Section 4 and if no attacker can forge their signature.

The algorithm provides both *safety* and *liveness* assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. Safety means that the replicated service satisfies linearizability [14] (modified to account for Byzantine-faulty clients [4]): it behaves like a centralized implementation that executes operations atomically one at a time. Safety requires the bound on the number of faulty replicas because a faulty replica can behave arbitrarily, e.g., it can destroy its state.

Safety is provided regardless of how many faulty clients are using the service (even if they collude with faulty replicas): all operations performed by faulty clients are observed in a consistent way by non-faulty clients. In particular, if the service operations are designed to preserve some invariants on the service state, faulty clients cannot break those invariants.

The safety property is insufficient to guard against faulty clients, e.g., in a file system a faulty client can write garbage data to some shared file. However, we limit the amount of damage a faulty client can do by providing access control: we authenticate clients and deny access if the client issuing a request does not have the right to invoke the operation. Also, services may provide operations to change the access permissions for a client. Since the algorithm ensures that the effects of access revocation operations are observed consistently by all clients, this provides a powerful mechanism to recover from attacks by faulty clients.

The algorithm does not rely on synchrony to provide safety. Therefore, it must rely on synchrony to provide liveness; otherwise it could be used to implement consensus in an asynchronous system, which is not possible [9]. We guarantee liveness, i.e., clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty and *delay*(t) does not grow faster than t indefinitely. Here, $delay(t)$ is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination (assuming the sender keeps retransmitting the message until it is received). (A more precise definition can be found in [4].) This is a rather weak synchrony assumption that is likely to be true in any real system provided network faults are eventually repaired, yet it enables us to circumvent the impossibility result in [9].

The resiliency of our algorithm is optimal: $3f+1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty (see [2] for a proof). This many replicas are needed because it must be possible to proceed after communicating with $n-f$ replicas, since f replicas might be faulty and not responding. However, it is possible that the f replicas that did not respond are not faulty and, therefore, f of those that responded might be faulty. Even so, there must still be enough responses that those from non-faulty replicas outnumber those from faulty ones, i.e., $n-2f>f$. Therefore $n>3f$.

The algorithm does not address the problem of fault-tolerant privacy: a faulty replica may leak information to an attacker. It is not feasible to offer fault-tolerant privacy in the general case because service operations may perform arbitrary computations using their arguments and the service state; replicas need this information in the clear to execute such operations efficiently. It is possible to use secret sharing schemes [35] to obtain privacy even in the presence of a threshold of malicious replicas [13] for the arguments and portions of the state that are opaque to the service operations. We plan to investigate these techniques in the future.

4 The Algorithm

Our algorithm is a form of *state machine* replication [17, 34]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. We denote the set of replicas by $\mathcal{R}$ and identify each replica using an integer in $\{0, ..., |\mathcal{R}|-1\}$. For simplicity, we assume $|\mathcal{R}| = 3f+1$ where f is the maximum number of replicas that may be faulty; although there could be more than $3f+1$ replicas, the additional replicas degrade performance (since more and bigger messages are being exchanged) without providing improved resiliency.

The replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. Views are numbered consecutively. The primary of a view is replica p such that $p = v \; mod \; |\mathcal{R}|$, where v is the view number. View changes are carried out when it appears that the primary has failed. Viewstamped Replication [26] and Paxos [18] used a similar approach to tolerate benign faults (as discussed in Section 8.)

The algorithm works roughly as follows:

1. A client sends a request to invoke a service operation to the primary
2. The primary multicasts the request to the backups
3. Replicas execute the request and send a reply to the client
4. The client waits for $f+1$ replies from different replicas with the same result; this is the result of the operation.

Like all state machine replication techniques [34], we impose two requirements on replicas: they must be *deterministic* (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result) and they must start in the same state. Given these two requirements, the algorithm ensures the safety property by guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures*.

The remainder of this section describes a simplified version of the algorithm. We omit discussion of how nodes recover from faults due to lack of space. We also omit details related to message retransmissions. Furthermore, we assume that message authentication is achieved using digital signatures rather than the more efficient scheme based on message authentication codes; Section 5 discusses this issue further. A detailed formalization of the algorithm using the I/O automaton model [21] is presented in [4].

4.1 The Client

A client c requests the execution of state machine operation o by sending a $\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}$ message to the primary. Timestamp t is used to ensure *exactly-once* semantics for the execution of client requests. Timestamps for c's requests are totally ordered such that later requests have higher timestamps than earlier ones; for example, the timestamp could be the value of the client's local clock when the request is issued.

Each message sent by the replicas to the client includes the current view number, allowing the client to track the view and hence the current primary. A client sends a request to what it believes is the current primary using a point-to-point message. The primary atomically multicasts the request to all the backups using the protocol described in the next section.

A replica sends the reply to the request directly to the client. The reply has the form $\langle \text{REPLY}, v, t, c, i, r\rangle_{\sigma_i}$ where v is the current view number, t is the timestamp of the corresponding request, i is the replica number, and r is the result of executing the requested operation.

The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r. This ensures that the result is valid, since at most f replicas can be faulty.

If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

In this paper we assume that the client waits for one request to complete before sending the next one. But we can allow a client to make asynchronous requests, yet preserve ordering constraints on them.

4.2 Normal-Case Operation

The state of each replica includes the state of the service, a *message log* containing messages the replica has accepted, and an integer denoting the replica's current view. We describe how to truncate the log in Section 4.3.

When the primary, p, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas. The primary starts the protocol immediately unless the number of messages for which the protocol is in progress exceeds a given maximum. In this case, it buffers the request. Buffered requests are multicast later as a group to cut down on message traffic and CPU overheads under heavy load; this optimization is similar to a group commit in transactional systems [11]. For simplicity, we ignore this optimization in the description below.

The three phases are *pre-prepare, prepare,* and *commit*. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views.

In the pre-prepare phase, the primary assigns a sequence number, n, to the request, multicasts a pre-prepare message with m piggybacked to all the backups, and appends the message to its log. The message has the form $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$, where v indicates the view in which the message is being sent, m is the client's request message, and d is m's digest.

Requests are not included in pre-prepare messages to keep them small. This is important because pre-prepare messages are used as a proof that the request was assigned sequence number n in view v in view changes. Additionally, it decouples the protocol to totally order requests from the protocol to transmit the request to the replicas; allowing us to use a transport optimized for small messages for protocol messages and a transport optimized for large messages for large requests.

A backup accepts a pre-prepare message provided:

- the signatures in the request and the pre-prepare message are correct and d is the digest for m;
- it is in view v;
- it has not accepted a pre-prepare message for view v and sequence number n containing a different digest;
- the sequence number in the pre-prepare message is between a low water mark, h, and a high water mark, H.

The last condition prevents a faulty primary from exhausting the space of sequence numbers by selecting a very large one. We discuss how H and h advance in Section 4.3.

If backup i accepts the $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$ message, it enters the *prepare* phase by multicasting a $\langle \text{PREPARE}, v, n, d, i\rangle_{\sigma_i}$ message to all other replicas and adds both messages to its log. Otherwise, it does nothing.

A replica (including the primary) accepts prepare messages and adds them to its log provided their signatures are correct, their view number equals the replica's current view, and their sequence number is between h and H.

We define the predicate *prepared*(m, v, n, i) to be true if and only if replica i has inserted in its log: the request m, a pre-prepare for m in view v with sequence number n, and $2f$ prepares from different backups that match the pre-prepare. The replicas verify whether the prepares match the pre-prepare by checking that they have the same view, sequence number, and digest.

The pre-prepare and prepare phases of the algorithm guarantee that non-faulty replicas agree on a total order for the requests within a view. More precisely, they ensure the following invariant: if *prepared*(m, v, n, i) is true then *prepared*(m', v, n, j) is false for any non-faulty replica j (including $i = j$) and any m' such that $D(m') \neq D(m)$. This is true because *prepared*(m, v, n, i) and $|\mathcal{R}| = 3f + 1$ imply that at least $f + 1$ non-faulty replicas have sent a pre-prepare or prepare for m in view v with sequence number n. Thus, for *prepared*(m', v, n, j) to be true at least one of these replicas needs to have sent two conflicting prepares (or pre-prepares if it is the primary for v), i.e., two prepares with the same view and sequence number and a different digest. But this is not possible because the replica is not faulty. Finally, our assumption about the strength of message digests ensures that the probability that $m \neq m'$ and $D(m) = D(m')$ is negligible.

Replica i multicasts a $\langle \text{COMMIT}, v, n, D(m), i\rangle_{\sigma_i}$ to the other replicas when *prepared*(m, v, n, i) becomes true. This starts the commit phase. Replicas accept commit messages and insert them in their log provided they are properly signed, the view number in the message is equal to the replica's current view, and the sequence number is between h and H

We define the *committed* and *committed-local* predicates as follows: *committed*(m, v, n) is true if and only if *prepared*(m, v, n, i) is true for all i in some set of $f+1$ non-faulty replicas; and *committed-local*(m, v, n, i) is true if and only if *prepared*(m, v, n, i) is true and i has accepted $2f + 1$ commits (possibly including its own) from different replicas that match the pre-prepare for m; a commit matches a pre-prepare if they have the same view, sequence number, and digest.

The commit phase ensures the following invariant: if *committed-local*(m, v, n, i) is true for some non-faulty i then *committed*(m, v, n) is true. This invariant and the view-change protocol described in Section 4.4 ensure that non-faulty replicas agree on the sequence numbers of requests that commit locally even if they commit in different views at each replica. Furthermore, it ensures that any request that commits locally at a non-faulty replica will commit at $f + 1$ or more non-faulty replicas eventually.

Each replica i executes the operation requested by m after *committed-local*(m, v, n, i) is true and i's state reflects the sequential execution of all requests with lower sequence numbers. This ensures that all non-faulty replicas execute requests in the same order as required to provide the safety property. After executing the requested operation, replicas send a reply to the client. Replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client to guarantee exactly-once semantics.

We do not rely on ordered message delivery, and therefore it is possible for a replica to commit requests out of order. This does not matter since it keeps the pre-prepare, prepare, and commit messages logged until the corresponding request can be executed.

Figure 1 shows the operation of the algorithm in the normal case of no primary faults. Replica 0 is the primary, replica 3 is faulty, and C is the client.

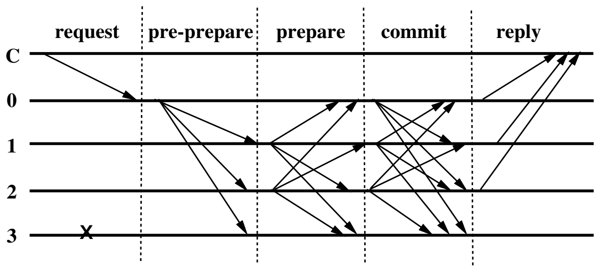

Figure 1: Normal Case Operation

4.3 Garbage Collection

This section discusses the mechanism used to discard messages from the log. For the safety condition to hold, messages must be kept in a replica's log until it knows that the requests they concern have been executed by at least $f + 1$ non-faulty replicas and it can prove this to others in view changes. In addition, if some replica misses messages that were discarded by all non-faulty replicas, it will need to be brought up to date by transferring all or a portion of the service state. Therefore, replicas also need some proof that the state is correct.

Generating these proofs after executing every operation would be expensive. Instead, they are generated periodically, when a request with a sequence number divisible by some constant (e.g., 100) is executed. We will refer to the states produced by the execution of these requests as *checkpoints* and we will say that a checkpoint with a proof is a *stable checkpoint*.

A replica maintains several logical copies of the service state: the last stable checkpoint, zero or more checkpoints that are not stable, and a current state. Copy-on-write techniques can be used to reduce the space overhead to store the extra copies of the state, as discussed in Section 6.3.

The proof of correctness for a checkpoint is generated as follows. When a replica i produces a checkpoint, it multicasts a message $\langle \text{CHECKPOINT}, n, d, i \rangle_{\sigma_i}$ to the other replicas, where n is the sequence number of the last request whose execution is reflected in the state and d is the digest of the state. Each replica collects checkpoint messages in its log until it has $2f + 1$ of them for sequence number n with the same digest d signed by different replicas (including possibly its own such message). These $2f + 1$ messages are the proof of correctness for the checkpoint.

A checkpoint with a proof becomes stable and the replica discards all pre-prepare, prepare, and commit messages with sequence number less than or equal to n from its log; it also discards all earlier checkpoints and checkpoint messages.

Computing the proofs is efficient because the digest can be computed using incremental cryptography [1] as discussed in Section 6.3, and proofs are generated rarely.

The checkpoint protocol is used to advance the low and high water marks (which limit what messages will be accepted). The low-water mark h is equal to the sequence number of the last stable checkpoint. The high water mark $H = h + k$, where k is big enough so that replicas do not stall waiting for a checkpoint to become stable. For example, if checkpoints are taken every 100 requests, k might be 200.

4.4 View Changes

The view-change protocol provides liveness by allowing the system to make progress when the primary fails. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is *waiting* for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v, the backup starts a view change to move the system to view $v+1$. It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a $\langle$VIEW-CHANGE$, v+1, n, \mathcal{C}, \mathcal{P}, i\rangle_{\sigma_i}$ message to all replicas. Here n is the sequence number of the last stable checkpoint s known to i, $\mathcal{C}$ is a set of $2f+1$ valid checkpoint messages proving the correctness of s, and $\mathcal{P}$ is a set containing a set $\mathcal{P}_m$ for each request m that prepared at i with a sequence number higher than n. Each set $\mathcal{P}_m$ contains a valid pre-prepare message (without the corresponding client message) and $2f$ matching, valid prepare messages signed by different backups with the same view, sequence number, and the digest of m.

When the primary p of view $v+1$ receives $2f$ valid view-change messages for view $v+1$ from other replicas, it multicasts a $\langle$NEW-VIEW$, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_p}$ message to all other replicas, where $\mathcal{V}$ is a set containing the valid view-change messages received by the primary plus the view-change message for $v+1$ the primary sent (or would have sent), and $\mathcal{O}$ is a set of pre-prepare messages (without the piggybacked request). $\mathcal{O}$ is computed as follows:

1. The primary determines the sequence number *min-s* of the latest stable checkpoint in $\mathcal{V}$ and the highest sequence number *max-s* in a prepare message in $\mathcal{V}$.
2. The primary creates a new pre-prepare message for view $v+1$ for each sequence number n between *min-s* and *max-s*. There are two cases: (1) there is at least one set in the $\mathcal{P}$ component of some view-change message in $\mathcal{V}$ with sequence number n, or (2) there is no such set. In the first case, the primary creates a new message $\langle$PRE-PREPARE$, v+1, n, d\rangle_{\sigma_p}$, where d is the request digest in the pre-prepare message for sequence number n with the highest view number in $\mathcal{V}$. In the second case, it creates a new pre-prepare message $\langle$PRE-PREPARE$, v+1, n, d^{null}\rangle_{\sigma_p}$, where d^{null} is the digest of a special *null* request; a null request goes through the protocol like other requests, but its execution is a no-op. (Paxos [18] used a similar technique to fill in gaps.)

Next the primary appends the messages in $\mathcal{O}$ to its log. If *min-s* is greater than the sequence number of its latest stable checkpoint, the primary also inserts the proof of stability for the checkpoint with sequence number *min-s* in its log, and discards information from the log as discussed in Section 4.3. Then it *enters* view $v+1$: at this point it is able to accept messages for view $v+1$.

A backup accepts a new-view message for view $v+1$ if it is signed properly, if the view-change messages it contains are valid for view $v+1$, and if the set $\mathcal{O}$ is correct; it verifies the correctness of $\mathcal{O}$ by performing a computation similar to the one used by the primary to create $\mathcal{O}$. Then it adds the new information to its log as described for the primary, multicasts a prepare for each message in $\mathcal{O}$ to all the other replicas, adds these prepares to its log, and enters view $v+1$.

Thereafter, the protocol proceeds as described in Section 4.2. Replicas redo the protocol for messages between *min-s* and *max-s* but they avoid re-executing client requests (by using their stored information about the last reply sent to each client).

A replica may be missing some request message m or a stable checkpoint (since these are not sent in new-view messages.) It can obtain missing information from another replica. For example, replica i can obtain a missing checkpoint state s from one of the replicas whose checkpoint messages certified its correctness in $\mathcal{V}$. Since $f+1$ of those replicas are correct, replica i will always obtain s or a later certified stable checkpoint. We can avoid sending the entire checkpoint by partitioning the state and stamping each partition with the sequence number of the last request that modified it. To bring a replica up to date, it is only necessary to send it the partitions where it is out of date, rather than the whole checkpoint.

4.5 Correctness

This section sketches the proof that the algorithm provides safety and liveness; details can be found in [4].

4.5.1 Safety

As discussed earlier, the algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally.

In Section 4.2, we showed that if *prepared*(m, v, n, i) is true, *prepared*(m', v, n, j) is false for any non-faulty replica j (including $i = j$) and any m' such that $D(m') \neq D(m)$. This implies that two non-faulty replicas agree on the sequence number of requests that commit locally in the same view at the two replicas.

The view-change protocol ensures that non-faulty replicas also agree on the sequence number of requests that commit locally in different views at different replicas. A request m commits locally at a non-faulty replica with sequence number n in view v only if *committed*(m, v, n) is true. This means that there is a set R_1 containing at least $f+1$ non-faulty replicas such that *prepared*(m, v, n, i) is true for every replica i in the set.

Non-faulty replicas will not accept a pre-prepare for view $v' > v$ without having received a new-view message for v' (since only at that point do they enter the view). But any correct new-view message for view $v' > v$ contains correct view-change messages from every replica i in a set R_2 of $2f + 1$ replicas. Since there are $3f + 1$ replicas, R_1 and R_2 must intersect in at least one replica k that is not faulty. k's view-change message will ensure that the fact that m prepared in a previous view is propagated to subsequent views, unless the new-view message contains a view-change message with a stable checkpoint with a sequence number higher than n. In the first case, the algorithm redoes the three phases of the atomic multicast protocol for m with the same sequence number n and the new view number. This is important because it prevents any different request that was assigned the sequence number n in a previous view from ever committing. In the second case no replica in the new view will accept any message with sequence number lower than n. In either case, the replicas will agree on the request that commits locally with sequence number n.

4.5.2 Liveness

To provide liveness, replicas must move to a new view if they are unable to execute a request. But it is important to maximize the period of time when at least $2f + 1$ non-faulty replicas are in the same view, and to ensure that this period of time increases exponentially until some requested operation executes. We achieve these goals by three means.

First, to avoid starting a view change too soon, a replica that multicasts a view-change message for view $v + 1$ waits for $2f + 1$ view-change messages for view $v + 1$ and then starts its timer to expire after some time T. If the timer expires before it receives a valid new-view message for $v + 1$ or before it executes a request in the new view that it had not executed previously, it starts the view change for view $v + 2$ but this time it will wait $2T$ before starting a view change for view $v + 3$.

Second, if a replica receives a set of $f + 1$ valid view-change messages from other replicas for views greater than its current view, it sends a view-change message for the smallest view in the set, even if its timer has not expired; this prevents it from starting the next view change too late.

Third, faulty replicas are unable to impede progress by forcing frequent view changes. A faulty replica cannot cause a view change by sending a view-change message, because a view change will happen only if at least $f + 1$ replicas send view-change messages, but it can cause a view change when it is the primary (by not sending messages or sending bad messages). However, because the primary of view v is the replica p such that $p = v \ mod \ |\mathcal{R}|$, the primary cannot be faulty for more than f consecutive views.

These three techniques guarantee liveness unless message delays grow faster than the timeout period indefinitely, which is unlikely in a real system.

4.6 Non-Determinism

State machine replicas must be deterministic but many services involve some form of non-determinism. For example, the time-last-modified in NFS is set by reading the server's local clock; if this were done independently at each replica, the states of non-faulty replicas would diverge. Therefore, some mechanism to ensure that all replicas select the same value is needed. In general, the client cannot select the value because it does not have enough information; for example, it does not know how its request will be ordered relative to concurrent requests by other clients. Instead, the primary needs to select the value either independently or based on values provided by the backups.

If the primary selects the non-deterministic value independently, it concatenates the value with the associated request and executes the three phase protocol to ensure that non-faulty replicas agree on a sequence number for the request and value. This prevents a faulty primary from causing replica state to diverge by sending different values to different replicas. However, a faulty primary might send the same, incorrect, value to all replicas. Therefore, replicas must be able to decide deterministically whether the value is correct (and what to do if it is not) based only on the service state.

This protocol is adequate for most services (including NFS) but occasionally replicas must participate in selecting the value to satisfy a service's specification. This can be accomplished by adding an extra phase to the protocol: the primary obtains authenticated values proposed by the backups, concatenates $2f + 1$ of them with the associated request, and starts the three phase protocol for the concatenated message. Replicas choose the value by a deterministic computation on the $2f + 1$ values and their state, e.g., taking the median. The extra phase can be optimized away in the common case. For example, if replicas need a value that is "close enough" to that of their local clock, the extra phase can be avoided when their clocks are synchronized within some delta.

5 Optimizations

This section describes some optimizations that improve the performance of the algorithm during normal-case operation. All the optimizations preserve the liveness and safety properties.

5.1 Reducing Communication

We use three optimizations to reduce the cost of communication. The first avoids sending most large replies. A client request designates a replica to send the result; all other replicas send replies containing just the digest of the result. The digests allow the client to check the correctness of the result while reducing network bandwidth consumption and CPU overhead significantly for large replies. If the client does not receive a correct result from the designated replica, it retransmits the request as usual, requesting all replicas to send full replies.

The second optimization reduces the number of message delays for an operation invocation from 5 to 4. Replicas execute a request *tentatively* as soon as the prepared predicate holds for the request, their state reflects the execution of all requests with lower sequence number, and these requests are all known to have committed. After executing the request, the replicas send tentative replies to the client. The client waits for $2f + 1$ matching tentative replies. If it receives this many, the request is guaranteed to commit eventually. Otherwise, the client retransmits the request and waits for $f + 1$ non-tentative replies.

A request that has executed tentatively may abort if there is a view change and it is replaced by a *null* request. In this case the replica reverts its state to the last stable checkpoint in the new-view message or to its last checkpointed state (depending on which one has the higher sequence number).

The third optimization improves the performance of read-only operations that do not modify the service state. A client multicasts a read-only request to all replicas. Replicas execute the request immediately in their tentative state after checking that the request is properly authenticated, that the client has access, and that the request is in fact read-only. They send the reply only after all requests reflected in the tentative state have committed; this is necessary to prevent the client from observing uncommitted state. The client waits for $2f+1$ replies from different replicas with the same result. The client may be unable to collect $2f+1$ such replies if there are concurrent writes to data that affect the result; in this case, it retransmits the request as a regular read-write request after its retransmission timer expires.

5.2 Cryptography

In Section 4, we described an algorithm that uses digital signatures to authenticate all messages. However, we actually use digital signatures only for view-change and new-view messages, which are sent rarely, and authenticate all other messages using message authentication codes (MACs). This eliminates the main performance bottleneck in previous systems [29, 22].

However, MACs have a fundamental limitation relative to digital signatures — the inability to prove that a message is authentic to a third party. The algorithm in Section 4 and previous Byzantine-fault-tolerant algorithms [31, 16] for state machine replication rely on the extra power of digital signatures. We modified our algorithm to circumvent the problem by taking advantage of specific invariants, e.g, the invariant that no two different requests prepare with the same view and sequence number at two non-faulty replicas. The modified algorithm is described in [5]. Here we sketch the main implications of using MACs.

MACs can be computed three orders of magnitude faster than digital signatures. For example, a 200MHz Pentium Pro takes 43ms to generate a 1024-bit modulus RSA signature of an MD5 digest and 0.6ms to verify the signature [37], whereas it takes only 10.3μs to compute the MAC of a 64-byte message on the same hardware in our implementation. There are other public-key cryptosystems that generate signatures faster, e.g., elliptic curve public-key cryptosystems, but signature verification is slower [37] and in our algorithm each signature is verified many times.

Each node (including active clients) shares a 16-byte secret session key with each replica. We compute message authentication codes by applying MD5 to the concatenation of the message with the secret key. Rather than using the 16 bytes of the final MD5 digest, we use only the 10 least significant bytes. This truncation has the obvious advantage of reducing the size of MACs and it also improves their resilience to certain attacks [27]. This is a variant of the secret suffix method [36], which is secure as long as MD5 is collision resistant [27, 8].

The digital signature in a reply message is replaced by a single MAC, which is sufficient because these messages have a single intended recipient. The signatures in all other messages (including client requests but excluding view changes) are replaced by vectors of MACs that we call *authenticators*. An authenticator has an entry for every replica other than the sender; each entry is the MAC computed with the key shared by the sender and the replica corresponding to the entry.

The time to verify an authenticator is constant but the time to generate one grows linearly with the number of replicas. This is not a problem because we do not expect to have a large number of replicas and there is a huge performance gap between MAC and digital signature computation. Furthermore, we compute authenticators efficiently; MD5 is applied to the message once and the resulting context is used to compute each vector entry by applying MD5 to the corresponding session key. For example, in a system with 37 replicas (i.e., a system that can tolerate 12 simultaneous faults) an authenticator can still be computed much more than two orders of magnitude faster than a 1024-bit modulus RSA signature.

The size of authenticators grows linearly with the number of replicas but it grows slowly: it is equal to $30 \times \lfloor \frac{n-1}{3} \rfloor$ bytes. An authenticator is smaller than an RSA signature with a 1024-bit modulus for $n \leq 13$ (i.e., systems that can tolerate up to 4 simultaneous faults), which we expect to be true in most configurations.

6 Implementation

This section describes our implementation. First we discuss the replication library, which can be used as a basis for any replicated service. In Section 6.2 we describe how we implemented a replicated NFS on top of the replication library. Then we describe how we maintain checkpoints and compute checkpoint digests efficiently.

6.1 The Replication Library

The client interface to the replication library consists of a single procedure, *invoke*, with one argument, an input buffer containing a request to invoke a state machine operation. The *invoke* procedure uses our protocol to execute the requested operation at the replicas and select the correct reply from among the replies of the individual replicas. It returns a pointer to a buffer containing the operation result.

On the server side, the replication code makes a number of upcalls to procedures that the server part of the application must implement. There are procedures to execute requests (*execute*), to maintain checkpoints of the service state (*make_checkpoint*, *delete_checkpoint*), to obtain the digest of a specified checkpoint (*get_digest*), and to obtain missing information (*get_checkpoint*, *set_checkpoint*). The *execute* procedure receives as input a buffer containing the requested operation, executes the operation, and places the result in an output buffer. The other procedures are discussed further in Sections 6.3 and 6.4.

Point-to-point communication between nodes is implemented using UDP, and multicast to the group of replicas is implemented using UDP over IP multicast [7]. There is a single IP multicast group for each service, which contains all the replicas. These communication protocols are unreliable; they may duplicate or lose messages or deliver them out of order.

The algorithm tolerates out-of-order delivery and rejects duplicates. View changes can be used to recover from lost messages, but this is expensive and therefore it is important to perform retransmissions. During normal operation recovery from lost messages is driven by the receiver: backups send negative acknowledgments to the primary when they are out of date and the primary retransmits pre-prepare messages after a long timeout. A reply to a negative acknowledgment may include both a portion of a stable checkpoint and missing messages. During view changes, replicas retransmit view-change messages until they receive a matching new-view message or they move on to a later view.

The replication library does not implement view changes or retransmissions at present. This does not compromise the accuracy of the results given in Section 7 because the rest of the algorithm is completely implemented (including the manipulation of the timers that trigger view changes) and because we have formalized the complete algorithm and proved its correctness [4].

6.2 BFS: A Byzantine-Fault-tolerant File System

We implemented BFS, a Byzantine-fault-tolerant NFS service, using the replication library. Figure 2 shows the architecture of BFS. We opted not to modify the kernel NFS client and server because we did not have the sources for the Digital Unix kernel.

A file system exported by the fault-tolerant NFS service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level *relay* processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the *invoke* procedure of our replication library, and sends the result back to the NFS client.

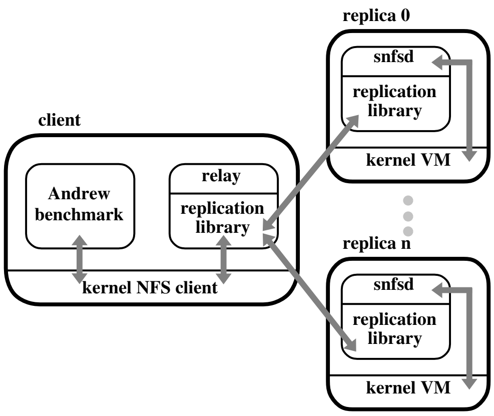

Figure 2: Replicated File System Architecture.

Each replica runs a user-level process with the replication library and our NFS V2 daemon, which we will refer to as *snfsd* (for simple *nfsd*). The replication library receives requests from the relay, interacts with *snfsd* by making upcalls, and packages NFS replies into replication protocol replies that it sends to the relay.

We implemented *snfsd* using a fixed-size memory-mapped file. All the file system data structures, e.g., inodes, blocks and their free lists, are in the mapped file. We rely on the operating system to manage the cache of memory-mapped file pages and to write modified pages to disk asynchronously. The current implementation uses 8KB blocks and inodes contain the NFS status information plus 256 bytes of data, which is used to store directory entries in directories, pointers to blocks in files, and text in symbolic links. Directories and files may also use indirect blocks in a way similar to Unix.

Our implementation ensures that all state machine replicas start in the same initial state and are deterministic, which are necessary conditions for the correctness of a service implemented using our protocol. The primary proposes the values for time-last-modified and time-last-accessed, and replicas select the larger of the proposed value and one greater than the maximum of all values selected for earlier requests. We do not require synchronous writes to implement NFS V2 protocol semantics because BFS achieves stability of modified data and meta-data through replication [20].

6.3 Maintaining Checkpoints

This section describes how *snfsd* maintains checkpoints of the file system state. Recall that each replica maintains several logical copies of the state: the current state, some number of checkpoints that are not yet stable, and the last stable checkpoint.

snfsd executes file system operations directly in the memory mapped file to preserve locality, and it uses copy-on-write to reduce the space and time overhead associated with maintaining checkpoints. *snfsd* maintains a copy-on-write bit for every 512-byte block in the memory mapped file. When the replication code invokes the *make_checkpoint* upcall, *snfsd* sets all the copy-on-write bits and creates a (volatile) checkpoint record, containing the current sequence number, which it receives as an argument to the upcall, and a list of blocks. This list contains the copies of the blocks that were modified since the checkpoint was taken, and therefore, it is initially empty. The record also contains the digest of the current state; we discuss how the digest is computed in Section 6.4.

When a block of the memory mapped file is modified while executing a client request, *snfsd* checks the copy-on-write bit for the block and, if it is set, stores the block's current contents and its identifier in the checkpoint record for the last checkpoint. Then, it overwrites the block with its new value and resets its copy-on-write bit. *snfsd* retains a checkpoint record until told to discard it via a *delete_checkpoint* upcall, which is made by the replication code when a later checkpoint becomes stable.

If the replication code requires a checkpoint to send to another replica, it calls the *get_checkpoint* upcall. To obtain the value for a block, *snfsd* first searches for the block in the checkpoint record of the stable checkpoint, and then searches the checkpoint records of any later checkpoints. If the block is not in any checkpoint record, it returns the value from the current state.

The use of the copy-on-write technique and the fact that we keep at most 2 checkpoints ensure that the space and time overheads of keeping several logical copies of the state are low. For example, in the Andrew benchmark experiments described in Section 7, the average checkpoint record size is only 182 blocks with a maximum of 500.

6.4 Computing Checkpoint Digests

snfsd computes a digest of a checkpoint state as part of a *make_checkpoint* upcall. Although checkpoints are only taken occasionally, it is important to compute the state digest incrementally because the state may be large. *snfsd* uses an incremental collision-resistant one-way hash function called AdHash [1]. This function divides the state into fixed-size blocks and uses some other hash function (e.g., MD5) to compute the digest of the string obtained by concatenating the block index with the block value for each block. The digest of the state is the sum of the digests of the blocks modulo some large integer. In our current implementation, we use the 512-byte blocks from the copy-on-write technique and compute their digest using MD5.

To compute the digest for the state incrementally, *snfsd* maintains a table with a hash value for each 512-byte block. This hash value is obtained by applying MD5 to the block index concatenated with the block value at the time of the last checkpoint. When *make_checkpoint* is called, *snfsd* obtains the digest d for the previous checkpoint state (from the associated checkpoint record). It computes new hash values for each block whose copy-on-write bit is reset by applying MD5 to the block index concatenated with the current block value. Then, it adds the new hash value to d, subtracts the old hash value from d, and updates the table to contain the new hash value. This process is efficient provided the number of modified blocks is small; as mentioned above, on average 182 blocks are modified per checkpoint for the Andrew benchmark.

7 Performance Evaluation

This section evaluates the performance of our system using two benchmarks: a micro-benchmark and the Andrew benchmark [15]. The micro-benchmark provides a service-independent evaluation of the performance of the replication library; it measures the latency to invoke a null operation, i.e., an operation that does nothing.

The Andrew benchmark is used to compare BFS with two other file systems: one is the NFS V2 implementation in Digital Unix, and the other is identical to BFS except without replication. The first comparison demonstrates that our system is practical by showing that its latency is similar to the latency of a commercial system that is used daily by many users. The second comparison allows us to evaluate the overhead of our algorithm accurately within an implementation of a real service.

7.1 Experimental Setup

The experiments measure normal-case behavior (i.e., there are no view changes), because this is the behavior that determines the performance of the system. All experiments ran with one client running two relay processes, and four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. The replicas and the client ran on identical DEC 3000/400 Alpha workstations. These workstations have a 133 MHz Alpha 21064 processor, 128 MB of memory, and run Digital Unix version 4.0. The file system was stored by each replica on a DEC RZ26 disk. All the workstations were connected by a 10Mbit/s switched Ethernet and had DEC LANCE Ethernet interfaces. The switch was a DEC EtherWORKS 8T/TX. The experiments were run on an isolated network.

The interval between checkpoints was 128 requests, which causes garbage collection to occur several times in any of the experiments. The maximum sequence number accepted by replicas in pre-prepare messages was 256 plus the sequence number of the last stable checkpoint.

7.2 Micro-Benchmark

The micro-benchmark measures the latency to invoke a null operation. It evaluates the performance of two implementations of a simple service with no state that implements null operations with arguments and results of different sizes. The first implementation is replicated using our library and the second is unreplicated and uses UDP directly. Table 1 reports the response times measured at the client for both read-only and read-write operations. They were obtained by timing 10,000 operation invocations in three separate runs and we report the median value of the three runs. The maximum deviation from the median was always below 0.3% of the reported value. We denote each operation by *a/b*, where *a* and *b* are the sizes of the operation argument and result in KBytes.

arg./res. (KB)	replicated		without replication
	read-write	read-only	
0/0	3.35 (309%)	1.62 (98%)	0.82
4/0	14.19 (207%)	6.98 (51%)	4.62
0/4	8.01 (72%)	5.94 (27%)	4.66

Table 1: Micro-benchmark results (in milliseconds); the percentage overhead is relative to the unreplicated case.

The overhead introduced by the replication library is due to extra computation and communication. For example, the computation overhead for the read-write 0/0 operation is approximately 1.06ms, which includes 0.55ms spent executing cryptographic operations. The remaining 1.47ms of overhead are due to extra communication; the replication library introduces an extra message round-trip, it sends larger messages, and it increases the number of messages received by each node relative to the service without replication.

The overhead for read-only operations is significantly lower because the optimization discussed in Section 5.1 reduces both computation and communication overheads. For example, the computation overhead for the read-only 0/0 operation is approximately 0.43ms, which includes 0.23ms spent executing cryptographic operations, and the communication overhead is only 0.37ms because the protocol to execute read-only operations uses a single round-trip.

Table 1 shows that the relative overhead is lower for the 4/0 and 0/4 operations. This is because a significant fraction of the overhead introduced by the replication library is independent of the size of operation arguments and results. For example, in the read-write 0/4 operation, the large message (the reply) goes over the network only once (as discussed in Section 5.1) and only the cryptographic overhead to process the reply message is increased. The overhead is higher for the read-write 4/0 operation because the large message (the request) goes over the network twice and increases the cryptographic overhead for processing both request and pre-prepare messages.

It is important to note that this micro-benchmark represents the worst case overhead for our algorithm because the operations perform no work and the unreplicated server provides very weak guarantees. Most services will require stronger guarantees, e.g., authenticated connections, and the overhead introduced by our algorithm relative to a server that implements these guarantees will be lower. For example, the overhead of the replication library relative to a version of the unreplicated service that uses MACs for authentication is only 243% for the read-write 0/0 operation and 4% for the read-only 4/0 operation.

We can estimate a rough lower bound on the performance gain afforded by our algorithm relative to Rampart [30]. Reiter reports that Rampart has a latency of 45ms for a multi-RPC of a null message in a 10 Mbit/s Ethernet network of 4 SparcStation 10s [30]. The multi-RPC is sufficient for the primary to invoke a state machine operation but for an arbitrary client to invoke an operation it would be necessary to add an extra message delay and an extra RSA signature and verification to authenticate the client; this would lead to a latency of at least 65ms (using the RSA timings reported in [29].) Even if we divide this latency by 1.7, the ratio of the SPECint92 ratings of the DEC 3000/400 and the SparcStation 10, our algorithm still reduces the latency to invoke the read-write and read-only 0/0 operations by factors of more than 10 and 20, respectively. Note that this scaling is conservative because the network accounts for a significant fraction of Rampart's latency [29] and Rampart's results were obtained using 300-bit modulus RSA signatures, which are not considered secure today unless the keys used to generate them are refreshed very frequently.

There are no published performance numbers for SecureRing [16] but it would be slower than Rampart because its algorithm has more message delays and signature operations in the critical path.

7.3 Andrew Benchmark

The Andrew benchmark [15] emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files.

We use the Andrew benchmark to compare BFS with two other file system configurations: NFS-std, which is the NFS V2 implementation in Digital Unix, and BFS-nr, which is identical to BFS but with no replication. BFS-nr ran two simple UDP relays on the client, and on the server it ran a thin veneer linked with a version of *snfsd* from which all the checkpoint management code was removed. This configuration does *not* write modified file system state to disk before replying to the client. Therefore, it does not implement NFS V2 protocol semantics, whereas both BFS and NFS-std do.

Out of the 18 operations in the NFS V2 protocol only `getattr` is read-only because the time-last-accessed attribute of files and directories is set by operations that would otherwise be read-only, e.g., `read` and `lookup`. The result is that our optimization for read-only operations can rarely be used. To show the impact of this optimization, we also ran the Andrew benchmark on a second version of BFS that modifies the `lookup` operation to be read-only. This modification violates strict Unix file system semantics but is unlikely to have adverse effects in practice.

For all configurations, the actual benchmark code ran at the client workstation using the standard NFS client implementation in the Digital Unix kernel with the same `mount` options. The most relevant of these options for the benchmark are: UDP transport, 4096-byte read and write buffers, allowing asynchronous client writes, and allowing attribute caching.

We report the mean of 10 runs of the benchmark for each configuration. The sample standard deviation for the total time to run the benchmark was always below 2.6% of the reported value but it was as high as 14% for the individual times of the first four phases. This high variance was also present in the NFS-std configuration. The estimated error for the reported mean was below 4.5% for the individual phases and 0.8% for the total.

Table 2 shows the results for BFS and BFS-nr. The comparison between BFS-strict and BFS-nr shows that the overhead of Byzantine fault tolerance for this service is low — BFS-strict takes only 26% more time to run the complete benchmark. The overhead is lower than what was observed for the micro-benchmarks because the client spends a significant fraction of the elapsed time computing between operations, i.e., between receiving the reply to an operation and issuing the next request, and operations at the server perform some computation. But the overhead is not uniform across the benchmark phases. The main reason for this is a variation in the amount of time the client spends computing between operations; the first two phases have a higher relative overhead because the client spends approximately 40% of the total time computing between operations, whereas it spends approximately 70% during the last three phases.

	BFS		
phase	strict	r/o lookup	BFS-nr
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

Table 2: Andrew benchmark: BFS vs BFS-nr. The times are in seconds.

The table shows that applying the read-only optimization to `lookup` improves the performance of BFS significantly and reduces the overhead relative to BFS-nr to 20%. This optimization has a significant impact in the first four phases because the time spent waiting for `lookup` operations to complete in BFS-strict is at least 20% of the elapsed time for these phases, whereas it is less than 5% of the elapsed time for the last phase.

	BFS		
phase	strict	r/o lookup	NFS-std
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

Table 3: Andrew benchmark: BFS vs NFS-std. The times are in seconds.

Table 3 shows the results for BFS vs NFS-std. These results show that BFS can be used in practice — BFS-strict takes only 3% more time to run the complete benchmark. Thus, one could replace the NFS V2 implementation in Digital Unix, which is used daily by many users, by BFS without affecting the latency perceived by those users. Furthermore, BFS with the read-only optimization for the `lookup` operation is actually 2% faster than NFS-std.

The overhead of BFS relative to NFS-std is not the same for all phases. Both versions of BFS are faster than NFS-std for phases 1, 2, and 5 but slower for the other phases. This is because during phases 1, 2, and 5 a large fraction (between 21% and 40%) of the operations issued by the client are *synchronous*, i.e., operations that require the NFS implementation to ensure stability of modified file system state before replying to the client. NFS-std achieves stability by writing modified state to disk whereas BFS achieves stability with lower latency using replication (as in Harp [20]). NFS-std is faster than BFS (and BFS-nr) in phases 3 and 4 because the client issues no synchronous operations during these phases.

8 Related Work

Most previous work on replication techniques ignored Byzantine faults or assumed a synchronous system model (e.g., [17, 26, 18, 34, 6, 10]). Viewstamped replication [26] and Paxos [18] use views with a primary and backups to tolerate benign faults in an asynchronous system. Tolerating Byzantine faults requires a much more complex protocol with cryptographic authentication, an extra pre-prepare phase, and a different technique to trigger view changes and select primaries. Furthermore, our system uses view changes only to select a new primary but never to select a different set of replicas to form the new view as in [26, 18].

Some agreement and consensus algorithms tolerate Byzantine faults in asynchronous systems (e.g,[2, 3, 24]). However, they do not provide a complete solution for state machine replication, and furthermore, most of them were designed to demonstrate theoretical feasibility and are too slow to be used in practice. Our algorithm during normal-case operation is similar to the Byzantine agreement algorithm in [2] but that algorithm is unable to survive primary failures.

The two systems that are most closely related to our work are Rampart [29, 30, 31, 22] and SecureRing [16]. They implement state machine replication but are more than an order of magnitude slower than our system and, most importantly, they rely on synchrony assumptions.

Both Rampart and SecureRing must exclude faulty replicas from the group to make progress (e.g., to remove a faulty primary and elect a new one), and to perform garbage collection. They rely on failure detectors to determine which replicas are faulty. However, failure detectors cannot be accurate in an asynchronous system [21], i.e., they may misclassify a replica as faulty. Since correctness requires that fewer than $1/3$ of group members be faulty, a misclassification can compromise correctness by removing a non-faulty replica from the group. This opens an avenue of attack: an attacker gains control over a single replica but does not change its behavior in any detectable way; then it slows correct replicas or the communication between them until enough are excluded from the group.

To reduce the probability of misclassification, failure detectors can be calibrated to delay classifying a replica as faulty. However, for the probability to be negligible the delay must be very large, which is undesirable. For example, if the primary has actually failed, the group will be unable to process client requests until the delay has expired. Our algorithm is not vulnerable to this problem because it never needs to exclude replicas from the group.

Phalanx [23, 25] applies quorum replication techniques [12] to achieve Byzantine fault-tolerance in asynchronous systems. This work does not provide generic state machine replication; instead, it offers a data repository with operations to read and write individual variables and to acquire locks. The semantics it provides for read and write operations are weaker than those offered by our algorithm; we can implement arbitrary operations that access any number of variables, whereas in Phalanx it would be necessary to acquire and release locks to execute such operations. There are no published performance numbers for Phalanx but we believe our algorithm is faster because it has fewer message delays in the critical path and because of our use of MACs rather than public key cryptography. The approach in Phalanx offers the potential for improved scalability; each operation is processed by only a subset of replicas. But this approach to scalability is expensive: it requires $n > 4f + 1$ to tolerate f faults; each replica needs a copy of the state; and the load on each replica decreases slowly with n (it is $O(1/\sqrt{n})$).

9 Conclusions

This paper has described a new state-machine replication algorithm that is able to tolerate Byzantine faults and can be used in practice: it is the first to work correctly in an asynchronous system like the Internet and it improves the performance of previous algorithms by more than an order of magnitude.

The paper also described BFS, a Byzantine-fault-tolerant implementation of NFS. BFS demonstrates that it is possible to use our algorithm to implement real services with performance close to that of an unreplicated service — the performance of BFS is only 3% worse than that of the standard NFS implementation in Digital Unix. This good performance is due to a number of important optimizations, including replacing public-key signatures by vectors of message authentication codes, reducing the size and number of messages, and the incremental checkpoint-management techniques.

One reason why Byzantine-fault-tolerant algorithms will be important in the future is that they can allow systems to continue to work correctly even when there are software errors. Not all errors are survivable; our approach cannot mask a software error that occurs at all replicas. However, it can mask errors that occur independently at different replicas, including nondeterministic software errors, which are the most problematic and persistent errors since they are the hardest to detect. In fact, we encountered such a software bug while running our system, and our algorithm was able to continue running correctly in spite of it.

There is still much work to do on improving our system. One problem of special interest is reducing the amount of resources required to implement our algorithm. The number of replicas can be reduced by using f replicas as witnesses that are involved in the protocol only when some full replica fails. We also believe that it is possible to reduce the number of copies of the state to $f + 1$ but the details remain to be worked out.

Acknowledgments

We would like to thank Atul Adya, Chandrasekhar Boyapati, Nancy Lynch, Sape Mullender, Andrew Myers, Liuba Shrira, and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology – Eurocrypt 97*, 1997.

[2] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4), 1995.

[3] R. Canneti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report #92-15, Computer Science Department, Hebrew University, 1992.

[4] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[5] M. Castro and B. Liskov. Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography. Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[6] F. Cristian, H. Aghili, H. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *International Conference on Fault Tolerant Computing*, 1985.

[7] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), 1990.

[8] H. Dobbertin. The Status of MD5 After a Recent Attack. *RSA Laboratories' CryptoBytes*, 2(2), 1996.

[9] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus With One Faulty Process. *Journal of the ACM*, 32(2), 1985.

[10] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for n >3t Processors in t+1 Rounds. *SIAM Journal of Computing*, 27(1), 1998.

[11] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *Database Engineering*, 8(2), 1985.

[12] D. Gifford. Weighted Voting for Replicated Data. In *Symposium on Operating Systems Principles*, 1979.

[13] M. Herlihy and J. Tygar. How to make replicated data secure. *Advances in Cryptology (LNCS 293)*, 1988.

[14] M. Herlihy and J. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, 1987.

[15] J. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.

[16] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Hawaii International Conference on System Sciences*, 1998.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), 1978.

[18] L. Lamport. The Part-Time Parliament. Technical Report 49, DEC Systems Research Center, 1989.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[20] B. Liskov et al. Replication in the Harp File System. In *ACM Symposium on Operating System Principles*, 1991.

[21] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[22] D. Malkhi and M. Reiter. A High-Throughput Secure Reliable Multicast Protocol. In *Computer Security Foundations Workshop*, 1996.

[23] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *ACM Symposium on Theory of Computing*, 1997.

[24] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Computer Security Foundations Workshop*, 1997.

[25] D. Malkhi and M. Reiter. Secure and Scalable Replication in Phalanx. In *IEEE Symposium on Reliable Distributed Systems*, 1998.

[26] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, 1988.

[27] B. Preneel and P. Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Crypto 95*, 1995.

[28] C. Pu, A. Black, C. Cowan, and J. Walpole. A Specialization Toolkit to Increase the Diversity of Operating Systems. In *ICMAS Workshop on Immunity-Based Systems*, 1996.

[29] M. Reiter. Secure Agreement Protocols. In *ACM Conference on Computer and Communication Security*, 1994.

[30] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.

[31] M. Reiter. A Secure Group Membership Protocol. *IEEE Transactions on Software Engineering*, 22(1), 1996.

[32] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC-1321, 1992.

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

[34] F. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

[35] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[36] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM Computer Communications Review*, 22(5), 1992.

[37] M. Wiener. Performance Comparison of Public-Key Cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.