

# Proactive Recovery in a Byzantine-Fault-Tolerant System

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

November 1st, 1999

## Abstract

This paper describes an asynchronous state-machine replication system that tolerates Byzantine faults caused by malicious attacks. Our system is the first to provide an algorithm to recover Byzantine-faulty replicas proactively, and it performs well because it does not use public-key cryptography for authentication. Whereas previous systems assumed less than  $1/3$  of the replicas were faulty during the lifetime of a system; ours can tolerate any number of faults provided less than  $1/3$  of the replicas become faulty within a window of vulnerability that is small under normal load conditions. The window may increase under a denial-of-service attack but we provide a secure mechanism to detect such attacks early. The paper presents the results of experiments that compare the performance of a replicated NFS built on top our system with an unreplicated NFS. These results suggest that the window of vulnerability can be made very small to achieve a high level of security with a low impact on service latency.

## 1 Introduction

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. Byzantine-fault-tolerant replication enables the implementation of robust services that continue to function correctly even when some of their replicas are compromised by an attacker.

This paper describes a new system for asynchronous state-machine replication [22, 36] that offers both integrity and high availability in the presence of Byzantine faults. Our system is interesting for two reasons: it improves the security of previous systems by recovering replicas proactively, and it performs well so that it can be used in practice to implement real services.

The best that previous asynchronous algorithms could guarantee was integrity for the service provided less than  $1/3$  of the replicas were faulty during the lifetime of a system. Our algorithm, however, recovers replicas proactively, so that it can tolerate any number of faults if less than  $1/3$  of the replicas become faulty within a window of vulnerability.

The window of vulnerability can be made very small under normal load conditions (e.g., a few minutes). Additionally, our algorithm provides *detection* of denial-of-service attacks aimed at increasing the window; replicas can time how long a recovery takes and alert their administrator if it exceeds some pre-established bound. Therefore, the integrity of the service cannot be compromised by an attack that escapes detection unless it compromises more than  $1/3$  of the replicas within a small window.

The window of vulnerability is unavoidable: since Byzantine-faulty replicas can discard the service state, we must bound the number of failures that can occur before recovery completes. However our algorithm is at least as robust to asynchrony as previous state-machine replication systems: it provides safety, regardless of the size of

---

This research was supported in part by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory, and in part by NEC.

the window of vulnerability, if less than  $1/3$  of the replicas fail during the lifetime of the system. This matches the safety guarantee provided by [8] and is stronger than the guarantees provided by other systems [32, 21].

Recovery has been widely studied in systems that tolerate benign failures (e.g., [24, 5]), but not in the presence of Byzantine faults. The latter problem is harder for three reasons. First, recovery must be proactive to prevent an attacker from compromising the service by corrupting  $1/3$  of the replicas slowly without being detected. Our algorithm addresses this problem by recovering replicas periodically independent of any failure detection mechanism. However, it is not possible to be sure whether a recovering replica is faulty. If it is not faulty, we need to make sure that it is not made faulty by the recovery algorithm. Otherwise, recovery could cause the number of faulty replicas to exceed the bound required to provide safety. In fact, we need to allow the replica to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery.

The second reason is that it is necessary to prevent an attacker from impersonating a replica that was faulty after it recovers. This is clearly a problem if the attacker learns the keys used to authenticate messages. But even assuming messages are authenticated using a secure cryptographic co-processor, it is unreasonable to believe that an attacker is unable to authenticate bad messages while it controls a faulty replica. These bad messages could be replayed later to compromise safety. To solve this problem, it is necessary to define a notion of authentication freshness so that replicas can reject messages that are not fresh. However, this leads to a further problem, since replicas may be unable to prove to a third party that some message they received is authentic (because it may no longer be fresh). All previous state-machine replication algorithms [8, 32, 21] relied on such proofs. Our algorithm does not, and it has the added advantage of enabling the use of symmetric cryptography for authentication of all protocol messages. This eliminates the use of public-key cryptography [31, 25, 21], the major performance bottleneck in previous systems.

Thirdly, efficient state transfer is harder in the presence of Byzantine faults and efficiency is crucial to enable frequent proactive recovery with low degradation of service performance. To bring a recovering replica up to date, the state transfer mechanism must check the local copy of the state to determine which portions are both up-to-date and not corrupt. Then, it must ensure that any missing state it obtains from other replicas is correct. We have developed an efficient hierarchical state transfer mechanism based on the hash chaining paradigm and incremental cryptography [2].

The system presented in [8] is the closest to ours. But that system did not provide recovery, it used a simpler view-change mechanism, and it used public-key cryptography to authenticate protocol messages. Rampart [31, 32, 33, 25] and SecureRing [21] provide membership protocols that allow re-integration of a server after it has recovered but do not address any of the three issues above.

Our algorithm has been implemented as a generic program library with a simple interface. This library can be used to provide Byzantine-fault-tolerant versions of different services. We used it to implement a Byzantine-fault-tolerant NFS file system [35]. The paper also describes experiments that compare the performance of our replicated NFS with an unreplicated NFS using the modified Andrew benchmark [19]. The results show that the latency of the replicated system without recovery is essentially identical to the latency of the unreplicated system. They also show that it is possible to recover replicas frequently to achieve a small window of vulnerability in the normal case (5 minutes) with little impact on service latency.

The rest of the paper is organized as follows. Section 2 presents our system model and lists our assumptions, Section 3 provides an overview of our approach, and Sections 5 to 8 describe our algorithm. Our implementation is described in Section 9 and some preliminary performance experiments are presented in Section 10. Section 11 discusses related work. Our conclusions are presented in Section 12.

## 2 System Model and Assumptions

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 restrictions mentioned below. We allow for a very strong adversary that can coordinate faulty nodes, delay communication, inject messages into the network, 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 use cryptographic techniques to establish session keys, authenticate messages, and produce digests. We use a Rabin-Williams public-key cryptosystem [39] with a 1024-bit modulus and the encoding schemes described in [3, 4] to establish 128-bit session keys. All messages are then authenticated using message authentication codes [38] computed using the session keys. The message digests are computed using MD5 [34].

We assume that the adversary (and the faulty nodes it controls) is computationally bound so that (with very high probability) it is unable to subvert these cryptographic techniques. For example, the adversary cannot produce a valid signature without knowing the private key, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties.

Previous Byzantine-fault tolerant state-machine replication systems [8, 32, 21] also rely on the assumptions described above. We require no additional assumptions to match the guarantees provided by these systems, i.e., to provide safety if less than  $1/3$  of the replicas become faulty during the lifetime of the system. But to tolerate more than  $1/3$  faulty replicas over the lifetime of the system we need additional assumptions.

At the very least we need to be able to mutually authenticate a faulty replica that recovers to the other replicas, and a reliable mechanism to trigger periodic recoveries. This could be achieved by involving system administrators in the recovery process, but such an approach is impractical given our goal of recovering replicas frequently. Instead, we rely on the following assumptions:

- **Secure Cryptography.** Each replica has a secure cryptographic co-processor, e.g., a Dallas Semiconductors iButton [11] or the security chip in the motherboard of the IBM PC 300PL [20]. The co-processor stores the replica's private key, and can sign and decrypt messages without exposing this key. It also contains a true random number generator, e.g., based on thermal noise, and a counter that never goes backwards. This enables it to append random numbers or the counter to messages it signs.
- **Fault-Tolerant Memory.** Each replica stores the public keys for other replicas in some memory that survives failures without being corrupted (provided the attacker does not have physical access to the machine). This memory could be a portion of the flash BIOS assuming that physical access to the machine is required to modify this portion of the BIOS (most motherboards can be configured this way).
- **Watchdog Timer.** Each replica has a *watchdog timer* that periodically interrupts processing and hands control to a *recovery monitor*, which is stored in the fault-tolerant memory. For this mechanism to be effective, an attacker should be unable to change the rate of watchdog interrupts without physical access to the machine. Some motherboards and extension cards offer the watchdog timer functionality but allow the timer to be reset without physical access to the machine. However, this is easy to fix by preventing write access to control registers unless some jumper switch is closed [10].

These assumptions are likely to hold when the attacker does not have physical access to the replicas, which we expect to be the common case. When they fail we can fall back on the system administrators to perform recovery.

Note that all previous proactive security algorithms [29, 17, 18, 30, 13] assume the entire program ran by a replica is in read-only memory so that it cannot be modified by an attacker, and most also assume that there are authenticated channels between the replicas. We could replace the *fault-tolerant memory* and *watchdog timer* assumptions by their first assumption, and the *secure cryptography* assumption could be replaced by their second assumption. However, their assumptions are less likely to hold in practice.

The only work on proactive security that does not assume authenticated channels is [30], but the best that a replica can do when its private key is compromised in their system is alert an administrator. Our *secure cryptography* assumption enables automatic recovery from most failures, and secure co-processors with the properties we require are now readily available, e.g., IBM is selling PCs with a cryptographic co-processor in the motherboard at essentially no added cost [20]. We also assume clients have a secure co-processor; this simplifies the description of the algorithm but it is not an intrinsic assumption.

### 3 Algorithm Properties

Our algorithm is a form of *state machine* replication [22, 36]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated

service with a *state* and some *operations*. The operations are not restricted to simple reads and writes of portions of the service state; they can perform arbitrary deterministic computations using the state and operation arguments.

The service is implemented by a set of replicas  $\mathcal{R}$  and each replica is identified using an integer in  $\{0, \dots, |\mathcal{R}|-1\}$ . Each replica maintains the service state and implements the service operations. For simplicity, we assume  $|\mathcal{R}| = 3f + 1$  where  $f$  is the maximum number of replicas that may be faulty. Service clients and replicas are non-faulty if they follow the algorithm and if no attacker can impersonate them (e.g., by forging their signatures).

Like all state machine replication techniques, 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. We can handle some common forms of non-determinism using the technique described in [8].

Our algorithm ensures safety for an execution provided at most  $f$  replicas become faulty within a window of vulnerability of size  $T_v = T_r + 2T_k$  and the assumptions in Section 2 hold. Safety means that the replicated service satisfies linearizability [16, 7]: it behaves like a centralized implementation that executes operations atomically one at a time.  $T_r$  is the maximum time between when a replica fails and when it recovers, and  $T_k$  is the maximum key refreshment period. We will discuss the period  $T_r$  further in Section 6.1. The additional  $2T_k$  time period shows up to account for message delays and replays (see Section 5.1). The values of  $T_v$ ,  $T_r$ , and  $T_k$  depend on the current execution and are unknown to the algorithm.

Our algorithm provides safety regardless of how many faulty clients are using the service (even if they collude with faulty replicas). In particular, if the service operations are designed to preserve some invariants on the service state, faulty clients cannot break those invariants. But clients can still damage the service state by violating consistency properties not enforced by the operations. We limit the amount of damage a faulty client can do by providing access control with revocable access.

The algorithm also guarantees liveness; non-faulty clients eventually receive replies to their requests provided: (1) at most  $f$  replicas become faulty within the window of vulnerability  $T_v$ , and (2) there is some (unknown) point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time  $d$  or all non-faulty clients have received replies to their requests. Here,  $d$  is a constant that depends on the timeout values used by the algorithm to refresh keys, and trigger view-changes and recoveries.

The resiliency of our algorithm is optimal [6]:  $3f + 1$  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.

## 4 Algorithm Overview

The algorithm works roughly as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. Therefore, since replicas are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client waits for  $f + 1$  replies from different replicas with the same result. Since at least one of these replicas is not faulty, this is the correct result of the operation.

The hard problem is guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures*. We use a primary-backup mechanism to achieve this. In such a mechanism, replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. We choose the primary of a view to be replica  $p$  such that  $p = v \bmod |\mathcal{R}|$ , where  $v$  is the view number and views are numbered consecutively.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request. But the primary may be faulty. Therefore, the backups trigger *view changes* when it appears that the primary has failed. View changes select a new primary to ensure liveness. Viewstamped Replication [28] and Paxos [23] used a similar approach to tolerate benign faults.

To tolerate Byzantine faults, every step taken by a node in our system is based on obtaining a *certificate*. A certificate is a set of messages certifying the same *statement* is correct and coming from different replicas. An example of a statement is: “the result value of the operation  $o$  requested by client  $c$  is  $r$ ”.

The size of the set of messages in a certificate is either  $f + 1$  or  $2f + 1$ , depending on the type of statement and

step being taken. The correctness of our system depends on a certificate never containing more than  $f$  messages sent by faulty replicas. Intuitively, a certificate of size  $f + 1$  is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size  $2f + 1$  ensures that it will also be possible to convince other replicas of the validity of the statement because non-faulty replicas remember the statements they certified.

Recovery complicates the construction of certificates; if a replica collects messages for a certificate over a sufficiently long period of time it can end up with more than  $f$  messages from faulty replicas. Our algorithm addresses this problem as follows: nodes refresh the keys that authenticate incoming messages periodically, and only accept *fresh* messages, i.e., messages authenticated using the keys created at the beginning of the current refreshment epoch.

However, this standard technique is insufficient to solve the problem. So the algorithm also forces certificates to contain only *equally fresh* messages, i.e., messages authenticated with keys created in the same refreshment phase. Nodes do this by discarding from their log any messages they received that are not part of a complete certificate when they refresh keys. This guarantees that non-faulty replicas never accept certificates with more than  $f$  bad messages in any execution that satisfies the following conditions: all replicas recover from a failure event after at most  $T_r$  time; all non-faulty replicas refresh their keys after at most  $T_k$  time; and there are no more than  $f$  failure events within  $T_r + T_k$ . We actually require a bigger window of vulnerability of size  $T_v = T_r + 2T_k$  because during view changes replicas may accept messages that are claimed authentic by  $f + 1$  replicas without directly checking their authentication token.

## 5 Normal Operation

This section explains how client requests are processed. We assume that all messages arrive at each replica; we discuss how to handle lost messages and lost state in Section 8. We actually provide a number of optimizations over what is described here including a scheme for executing read requests with just one message exchange.

### 5.1 Message Authentication

We use message authentication codes (MACs) to authenticate messages. There is a pair of session keys for each pair of replicas  $i$  and  $j$ :  $k_{i,j}$  is used to compute MACs for messages sent from  $i$  to  $j$ , and  $k_{j,i}$  is used for messages sent from  $j$  to  $i$ . We use a pair of keys rather than a single shared session key to allow replicas to change their keys independently. Each replica has in addition a single session key for each active client.

Some messages in the protocol contain a single MAC computed using the secret suffix method of [38]; we denote such a message as  $\langle m \rangle_{\mu_{i,j}}$ , where  $i$  is the sender  $j$  is the receiver and the MAC is computed using  $k_{i,j}$ . Other messages contain *authenticators*; we denote such a message as  $\langle m \rangle_{\alpha_i}$ , where  $i$  is the sender. An authenticator is a vector of MACs, one per replica  $j$  ( $j \neq i$ ), where the MAC in entry  $j$  is computed using  $k_{i,j}$ . The receiver of a message verifies its authenticity by checking the corresponding MAC in the authenticator.

Replicas and clients refresh the session keys used to send messages to them by sending *new-key* messages periodically (e.g., every minute). The same mechanism is used to establish the initial session keys. The message has the form  $\langle \text{NEW-KEY}, i, \dots, \{k_{j,i}\}_{\epsilon_j}, \dots, t \rangle_{\sigma_i}$ . The message is signed by the secure co-processor (using the replica's private key) and  $t$  is the value of its counter; the counter is incremented by the co-processor and appended to the message every time it generates a signature. (This prevents surpress-replay attacks [15].) Each  $k_{j,i}$  is the key replica  $j$  should use to authenticate messages it sends to  $i$  in the future;  $k_{j,i}$  is encrypted by  $j$ 's public key, so that only  $j$  can read it. Replicas use timestamp  $t$  to detect spurious new-key messages:  $t$  must be larger than the timestamp of the last new-key message received from  $i$ .

Each replica shares a single secret key with each client; this key is used for communication in both directions. The key is refreshed by the client periodically, using the new-key message. If a client neglects to do this within some system-defined period, a replica discards its current key for that client, which forces the client to refresh the key.

When a replica or client sends a new-key message, it discards all messages in its log that are not part of a complete certificate and it rejects any messages it receives in the future that are authenticated with old keys. This

is necessary to ensure that certificates contain only *equally fresh* messages (see Section 4).

## 5.2 The Client

A client  $c$  requests the execution of state machine operation  $o$  by sending a  $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}$  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.

The primary atomically multicasts the request to all the backups (using the protocol described below). Replicas execute the requested operation and send the reply directly to the client. The reply has the form  $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$  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. This message includes the current view number so that clients can track the current primary.

The client waits for  $f + 1$  equally fresh replies from different replicas, and with the same  $t$  and  $r$ , before accepting the result  $r$ . This certificate ensures that the result is valid.

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 and remembers the request. If the request is not executed, the primary will eventually be suspected to be faulty by enough replicas to cause a view change and select a new primary.

## 5.3 Processing Requests

The primary uses a three-phase protocol to atomically multicast requests to the replicas. 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. 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.

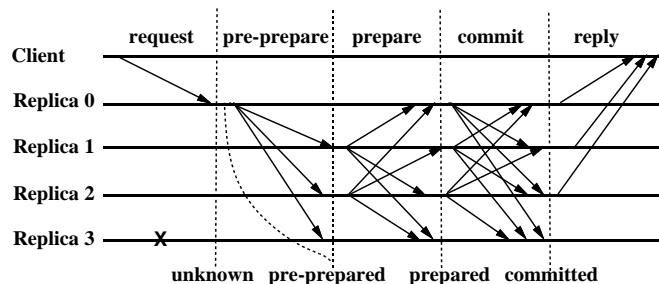


Figure 1: Normal Case Operation

Each replica stores the service state, a *log* containing information about requests, and an integer denoting the replica's current view. The log records information about the request associated with each sequence number, including its status; the possibilities are: *unknown* (the initial status), *pre-prepared*, *prepared*, and *committed*. Figure 1 also shows the evolution of the request status as the protocol progresses. We describe how to truncate the log in Section 5.4.

When the primary  $p$  receives a request  $m$  from a client, it starts the pre-prepare phase by assigning a sequence number  $n$  to  $m$ . Then, it multicasts a pre-prepare message to all the backups, and marks  $m$  as pre-prepared with sequence number  $n$ . The message has the form  $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\alpha_p}, m \rangle$ , where  $v$  indicates the view in which the message is being sent, and  $d$  is  $m$ 's digest.

Like pre-prepares, the prepare and commit messages sent in the other phases also contain  $n$  and  $v$ . A replica only accepts one of these messages provided it is in view  $v$ ; it can verify the authenticity of the message; and  $n$  is between a low water mark,  $h$ , and a high water mark,  $H$ . The last condition is necessary to enable garbage collection and prevent 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 5.4.

A backup  $i$  accepts the pre-prepare message provided (in addition to the conditions above): it has not accepted a pre-prepare for view  $v$  and sequence number  $n$  containing a different digest, it can verify the authenticity of  $m$ , and  $d$  is  $m$ 's digest. If  $i$  accepts the pre-prepare, it marks  $m$  as pre-prepared with sequence number  $n$ , and enters the *prepare* phase by multicasting a  $\langle \text{PREPARE}, v, n, d, i \rangle_{\alpha_i}$  message to all other replicas.

When replica  $i$  has accepted a pre-prepare message and  $2f$  equally-fresh prepare messages for the same sequence number  $n$  and digest  $d$ , each from a different replica (including itself), it marks the message as *prepared*. The protocol guarantees that other non-faulty replicas will either prepare the same request or will not prepare any request with sequence number  $n$  in view  $v$ .

Replica  $i$  multicasts  $\langle \text{COMMIT}, v, n, d, i \rangle_{\alpha_i}$  when the request prepares. This starts the commit phase. When a replica has accepted  $2f + 1$  equally-fresh commit messages for the same sequence number  $n$  and digest  $d$  from different replicas (including itself), it marks the message as *committed*. The protocol guarantees that the request is prepared with sequence number  $n$  in view  $v$  at  $f + 1$  or more non-faulty replicas. This enables information about committed request to be propagated to new views.

Replica  $i$  executes the operation requested by the client when  $m$  is committed with sequence number  $n$  and the replica has executed 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 replicas keep requests until they can be executed.

## 5.4 Garbage Collection

Replicas can discard entries from their log once they have been executed by at least  $f + 1$  non-faulty replicas; this many replicas are needed to ensure that the execution of that request will be known in a view change. We can determine this condition by extra communication, but to reduce cost we do the communication only when a request with a sequence number divisible by some constant  $K$  (e.g.,  $K = 100$ ) is executed. We will refer to the states produced by the execution of these requests as *checkpoints* and we will say that a checkpoint known at  $f + 1$  non-faulty replicas 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. We use copy-on-write techniques to reduce the space overhead to store the extra copies of the state, as discussed in Section 8.

When replica  $i$  produces a checkpoint, it multicasts a  $\langle \text{CHECKPOINT}, n, d, i \rangle_{\alpha_i}$  message 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. The digest can be computed efficiently using incremental cryptography [2] as discussed in Section 8.

Each replica waits until it has a certificate containing  $2f + 1$  valid checkpoint messages for sequence number  $n$  with the same digest  $d$ , all equally fresh and sent by different replicas (including possibly its own such message). At this point the checkpoint is known to be stable and the replica discards all entries in its log with sequence numbers less than or equal to  $n$ ; it also discards all earlier checkpoints.

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  $H = h + cK$ , where  $c$  is a small constant (e.g.,  $c = 2$ ) that is big enough so that replicas do not stall waiting for a checkpoint to become stable.

## 6 Recovery

Recovery starts when the watchdog timer goes off. The recovery monitor checkpoints the state of the service and the replication protocol to disk, computes a digest of all the code in the replica's machine (including the operating system, daemons, and configuration files), compares it with a prestored digest (which is kept in the fault-tolerant memory), and reloads the code if the two do not match. An alternative to checking the code is to store it on a read-only medium. For example, several modern disks, e.g, the Seagate Cheetah 18LP [37], can be write protected by physically closing a jumper switch.

Then the recovery monitor reboots the system and restarts the service in recovery mode from the checkpointed state. This ensures that the operating system code and data structures are restored to a correct state, and verify any invariants necessary for the correct execution of the recovery protocol. In particular, it prevents an attacker from leaving behind a Trojan horse.

At this point the replica's code is good but its state may be bad. The rest of recovery determines the true situation and restores the state if necessary. In particular, by the time recovery is complete we ensure that (1) an intruder will be unable to impersonate  $i$  to send bad messages to other replicas; and (2)  $i$  will not act on bad information in its state. However recovery must be done carefully so that a replica that was not faulty remains this way: it must retain its state and use it to process requests even while it is recovering. This is vital to ensure both safety and liveness if the recovering replica is non-faulty (which is likely to be the common case) since otherwise the recovering replica could become the  $f+1$ st fault.

The recovering node  $i$  starts by discarding its secret keys for clients, and the secret keys used by other replicas to authenticate messages sent to  $i$ . Then it multicasts a new-key message to all the replicas. This step is important if  $i$  was faulty because otherwise the attacker could prevent a successful recovery by impersonating any client or replica.

Next,  $i$  estimates  $ls$ , the maximum stable checkpoint sequence number at any non-faulty replica. Estimation works as follows:  $i$  multicasts a  $\langle \text{QUERY-STABLE}, i, r \rangle_{\alpha_i}$  message to all the other replicas, where  $r$  is a fresh random nonce. When replica  $j$  receives this message it replies  $\langle \text{REPLY-STABLE}, c, p, i, r \rangle_{\mu_{ji}}$ , where  $c$  and  $p$  are the sequence numbers of the last checkpoint at  $j$  and the last request prepared by  $j$  respectively.  $i$  keeps retransmitting the query message and processing replies; for each replica it keeps the minimum value of  $c$  and the maximum value of  $p$  it received in replies from that replica. It also keeps its own values of  $c$  and  $p$ . During estimation  $i$  does not handle any other protocol messages except new-key, query-stable, and status messages (see Section 8).

The recovering replica uses the responses to select the estimate  $ls'$  for  $ls$  as follows:  $ls'$  is a value  $c$  received from replica  $j$  such that  $2f$  replicas other than  $j$  (possibly including  $i$  itself) reported values for  $c$  less than or equal to  $ls'$  and  $f$  replicas other than  $j$  reported values of  $p$  greater than or equal to  $ls'$ . For safety we require that  $ls'$  be greater than or equal to  $ls$  because otherwise  $i$  might discard its state when it is non-faulty and thus become faulty. This is insured because  $ls$  is stable when estimation starts and, therefore, at least  $f + 1$  non-faulty replicas will have created the checkpoint with sequence number  $ls$ . The test against  $p$  ensures that  $ls'$  is close to a stable checkpoint at some non-faulty replica since at least one non-faulty replica reports a  $p$  not less than  $ls'$ ; this is important because it prevents a faulty replica from prolonging  $i$ 's recovery. Estimation is live because there are  $2f + 1$  non-faulty replicas and they only propose a value of  $c$  if the corresponding request committed and that implies that it prepared at least  $f + 1$  correct replicas.

After determining  $ls'$ , the replica discards its entire protocol state if it has any messages with sequence numbers greater than  $ls' + cK$ . Then it sends a recovery request with the form:  $\langle \text{REQUEST}, \langle \text{RECOVERY}, ls' \rangle, t, i \rangle_{\sigma_i}$ . This message is produced by the cryptographic co-processor and  $t$  is the co-processor's counter.

Replica  $j$  accepts the recovery request provided the value of  $t$  is greater than any value of  $t$  it received in an earlier recovery request from  $i$ , and it has not accepted a different recovery request from  $i$  recently (where recently can be defined as half of the watchdog timeout). This is important to prevent a denial-of-service attack where non-faulty replicas are kept busy executing recovery requests. If  $j$  accepts the recovery request, it sends its own new-key message. This will prevent it from receiving counterfeit messages sent by an attacker that is impersonating the recovering node. Otherwise, the recovery request is treated like any other request: it is assigned a sequence number by the primary and it goes through the usual three phases. Replicas also send a new-key message when they fetch missing state (see Section 8) and determine that it reflects the execution of a new recovery request.

The reply to the recovery request includes  $n_i$ , the sequence number at which it was executed;  $i$  uses the same

protocol as the client to collect the correct reply to its recovery request but waits for  $2f+1$  replies. Then it computes its *recovery point*,  $H = cK + \max(ls', \lfloor n_i/K \rfloor \times K)$ . It also computes a valid view (see Section 7); it retains its current view if there are  $f + 1$  replies for views greater than or equal to it, else it changes to the median of the views in the replies.  $i$  continues to participate in the protocol as if it were not recovering except that it will not send any messages above  $H$  until it has the correct stable checkpoint for that sequence number. While  $i$  is recovering, it needs to determine whether its state is correct, and if not it needs to discard the bad parts and fetch them from other replicas (by using the state transfer mechanism discussed in Section 8).

Replica  $i$  is *recovered* when the checkpoint with sequence number  $H$  is stable. This ensures that any state other replicas relied on  $i$  to have is actually held by  $f + 1$  non-faulty replicas. Therefore if some other replica fails now, we can be sure the state of the system will not be lost. This is true because the estimation procedure run at the beginning of recovery ensures that while recovering  $i$  never sends bad messages for sequence numbers above the recovery point. Furthermore, replicas only accept messages above the recovery point if their state reflects the execution of the recovery request; by then they will have sent the new-key messages, and therefore will not be fooled by an intruder impersonating  $i$ .

If clients aren't using the system this could delay recovery, since the system needs to reach request  $H$  before recovery occurs. However, this is easy to fix. While a recovery is occurring, the primary can speed things up by sending pre-prepares for special *null* requests. A null request goes through the protocol like other requests, but its execution is a no-op.

Our scheme has the nice property that any replica knows that  $i$  has completed its recovery when checkpoint  $H$  is stable. This allows replicas to estimate the duration of  $i$ 's recovery, which is useful to detect denial-of-service attacks that slow down recovery with low false positives, and to securely adjust the watchdog timeout.

## 6.1 Discussion

Our algorithm ensures safety for an execution  $\tau$  provided at most  $f$  replicas become faulty within a window of vulnerability of size  $T_v = 2T_k + T_r$ . The values of  $T_k$  and  $T_r$  are characteristic of each execution  $\tau$  and unknown to the algorithm.  $T_k$  is the maximum key refreshment period in  $\tau$  for a non-faulty replica, and  $T_r$  is the maximum time between when a replica fails and when it recovers in  $\tau$ ;  $T_r$  is greater than or equal to the maximum time between watchdog timeouts,  $T_w$ .

We have little control over the value of  $T_v$  because it may be increased by a denial-of-service attack, but we have good control over  $T_k$  and  $T_w$  because their values are determined by timer rates, which are quite stable. Setting these timeout values involves a tradeoff between security and performance: small values improve security by reducing the window of vulnerability but degrade performance by causing more frequent recoveries and key changes.

The results in Section 10 indicate that  $T_k$  can be small, e.g., 30 seconds, without impacting performance significantly. To provide liveness, however,  $T_k$  should be substantially larger than 3 message delays under normal load conditions.

The value of  $T_w$  should be set based on the time it takes to recover a non-faulty replica under normal load conditions,  $R_n$ . There is no point in recovering a replica when its previous recovery has not yet finished; and we stagger the recoveries so that no more than  $f$  replicas are recovering at once, since otherwise service could be interrupted even without an attack. Therefore, we set  $T_w = 4 \times s \times R_n$ . Here, the factor 4 accounts for the staggered recovery of  $3f + 1$  replicas  $f$  at a time, and  $s$  is a safety factor to account for benign overload conditions.

We expect  $R_n$  to be dominated by the time to reboot and check the correctness of the replica's copy of the service state. Since the replica checks its state without loading the network or any other replica, we expect the time to recover  $f$  replicas in parallel and the time to recover a replica under benign overload conditions to be close to  $R_n$ ; thus we can set  $s$  close to 1.

The results in Section 10 indicate that  $T_w$  can be set to a few minutes for services with state smaller 1GB. We are currently investigating ways of handling larger states, e.g., by using mostly read-only media so that the cost of state transfer can be related to the amount of state modified in some time period rather than the size of the state.

## 7 View Change Protocol

The view-change protocol provides liveness by allowing the system to make progress when the current primary fails. But the protocol must also preserve safety: it must ensure that non-faulty replicas agree on the sequence numbers of committed requests across views. Providing safety is hard because the certificates used by our protocol cannot be transferred between replicas. The protocol must also prevent intruders from using view changes as an attack mechanism; a technique to prevent this is presented in [8].

View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup starts a timer when it receives a request and the timer is not already running. The timer is stopped when the request executes, but restarted immediately if the backup has received some other request that has not yet executed. If the timer expires, the backup starts a view change to move the system to view  $v + 1$ . It enters the new view, but the view is pending rather than active; a replica does not accept any pre-prepare, prepare or commit messages in a pending view. If at least  $2f + 1$  replicas start a view change, the system changes to a new view and to a different primary. The primary for a view is the replica whose identifier is congruent to the view number modulo the number of replicas.

The view change algorithm must work even when more than one view change occurs before the system is able to continue normal operation. For this reason, replicas must remember what happened in earlier views. This information is maintained in two sets, the  $PSet$  and the  $QSet$ . Like the log, these sets contain information only for messages with numbers above the last stable checkpoint and below that checkpoint plus  $cK$ ; therefore only limited storage is required. A replica also stores the messages corresponding to the entries in these sets. When the system is running normally, these sets are empty.

The  $PSet$  at replica  $i$  stores information about messages that have prepared at  $i$  in the past, i.e., messages for which  $i$  sent a commit message in an earlier view. Its entries are tuples  $\langle n, d, v \rangle$  meaning that a request with digest  $d$  prepared at  $i$  with number  $n$  in view  $v$  and no request with a different digest prepared at  $i$  in a view greater than  $v$ . The  $QSet$  stores information about messages that have pre-prepared at  $i$ . Its entries are tuples  $\langle n, d, v, u \rangle$  meaning that  $v$  is the latest view in which a request pre-prepared with sequence number  $n$  and digest  $d$  at  $i$ ; no request with a different digest pre-prepared at  $i$  in a view greater than  $v$ ; and  $u$  is the latest view for which a request with digest different from  $d$  pre-prepared at  $i$ . The view-change protocol ensures that no request prepared globally with sequence number  $n$  in any view  $v' \leq u$ .

When  $i$  enters view  $v + 1$ , it multicasts a  $\langle \text{VIEW-CHANGE}, v + 1, ls, \mathcal{C}, \mathcal{P}, \mathcal{Q}, i \rangle_{\alpha_i}$  message to all replicas. Here  $ls$  is the sequence number of the latest stable checkpoint known to  $i$ ;  $\mathcal{C}$  is a set of pairs with the sequence number and digest of each checkpoint stored at  $i$ ; and  $\mathcal{P}$  and  $\mathcal{Q}$  are sets containing a tuple for every request that is prepared or pre-prepared, respectively, at  $i$ . These sets are computed using the information in the log, the  $PSet$ , and  $QSet$ , as explained in Figure 2. Once the view-change message has been sent,  $i$  stores  $\mathcal{P}$  in  $PSet$ ,  $\mathcal{Q}$  in  $QSet$ , and clears its log.

Let  $v$  be the current view (before the view change) and  $ls$  be the number of the current stable checkpoint

For all  $n$  s.t.  $ls < n \leq ls + cK$

if message number  $n$  with digest  $d$  is prepared or committed in the log add  $\langle n, d, v \rangle$  to  $\mathcal{P}$   
 else if there is an entry  $e$  for  $n$  in  $PSet$  add  $\langle n, e.d, e.v \rangle$  to  $\mathcal{P}$

if message number  $n$  with digest  $d$  is pre-prepared, prepared or committed in the log then

if there is an entry  $e$  for  $n$  in  $QSet$  then

if  $e.d = d$  then add  $\langle n, d, v, e.u \rangle$  to  $\mathcal{Q}$  else add  $\langle n, d, v, e.v \rangle$  to  $\mathcal{Q}$

else add  $\langle n, d, v, -1 \rangle$  to  $\mathcal{Q}$

else if there is an entry  $e$  for  $n$  in  $QSet$  add  $\langle n, e.d, e.v, e.u \rangle$  to  $\mathcal{Q}$

Figure 2: Computing  $\mathcal{P}$  and  $\mathcal{Q}$

All replicas collect view-change messages for  $v + 1$  and send acknowledgments for them to  $v + 1$ 's primary,  $p$ . The acknowledgment messages have the form  $\langle \text{VIEW-CHANGE-ACK}, v + 1, i, i', d' \rangle_{\mu_{ip}}$  where  $i$  is the identifier of the sending replica,  $d'$  is the digest of the view-change message being acknowledged, and  $i'$  is the replica that sent that

view-change message. These acknowledgements take care of the problem that certificates cannot be transferred between replicas as explained later in this section.

The new primary  $p$  collects view-change and view-change-ack messages (including messages from itself). It stores view-change messages in a set  $\mathcal{S}$ . It adds a view-change message received from replica  $i$  to  $\mathcal{S}$  when it has also received  $2f - 1$  view-change-acks for  $i$ 's view-change message from other replicas. Each entry in  $\mathcal{S}$  is for a different replica.

The primary needs to decide for each number above the most recent stable checkpoint whether a request with that number might have committed in a previous view, in which case it propagates a pre-prepare for it. If a request with that number was in progress but had not yet committed, the primary might still propagate a pre-prepare for it, or it might propagate a special *null* request that goes through the protocol as a regular request but whose execution is a no-op. The decision procedure used by the primary is sketched in Figure 3; this procedure runs only when  $\mathcal{S}$  has at least  $2f + 1$  view-change messages (including the primary's own view-change message); and it runs each time the primary receives new information, e.g., when it adds another view-change message to  $\mathcal{S}$ .

let  $CP$  be the selected checkpoint number. Initially it is 0.

1. if there is an entry in  $\mathcal{S}$  for some replica  $i$  that proposes a stable checkpoint number  $n$  with digest  $d$  s.t.  $n > CP$ ,  $2f$  other entries in  $\mathcal{S}$  propose stable checkpoint numbers that are less than or equal to  $n$ , and  $f$  other entries propose checkpoint number  $n$  with the same digest  $d$ , then update  $CP$  to be  $n$ .
2. let  $N = CP + cK$
3. for each  $n$  s.t.  $CP < n \leq N$ 
  - A. if there exists an entry  $m$  in  $\mathcal{S}$  s.t.  $m$ 's  $\mathcal{P}$  contains  $\langle n, d, v \rangle$  that verifies:
    - i. at least  $2f$  other entries in  $\mathcal{S}$  either have  $ls < n$  and no entry for  $n$  in their  $\mathcal{P}$ , or have an entry  $\langle n, v', d' \rangle$  in their  $\mathcal{P}$  with either  $v' < v$  or  $(v' = v \wedge d' = d)$
    - ii. at least  $f$  other entries in  $\mathcal{S}$  have  $\langle n, d, v', u' \rangle$  in their  $\mathcal{Q}$  with either  $(v' \geq v \wedge d' = d)$  or  $u \geq v$
    - iii. the primary itself has the request for  $n$  with digest  $d$
then the primary selects a pre-prepare for  $d$  for message number  $n$ .
  - B. Otherwise if there exists an entry  $m$  in  $\mathcal{S}$  s.t.  $m.ls < n$  and there is no entry for  $n$  in  $m$ 's  $\mathcal{P}$  and at least  $2f$  other entries in  $\mathcal{S}$  with  $ls < n$  also have no entry for  $n$  in their  $\mathcal{P}$  then the primary selects null for message number  $n$ .

Figure 3: Decision procedure at the primary.

We now argue informally that this procedure will select the correct value for each message number. If the request at  $n$  committed at some non-faulty replica, it prepared at at least  $f + 1$  non-faulty replicas and the view-change messages sent by those replica will list that request as prepared for that number. Any set of at least  $2f + 1$  view-change messages for the new view will have to include a message from one of these non-faulty replicas that prepared the request. Therefore, the primary for the new view will be unable to select a different request for that slot because no other request will be able to satisfy conditions 3A(i) and 3B.

The primary will also be able to make the right decision eventually: condition 3A(i) will be verified because there are  $2f + 1$  non-faulty replicas and non-faulty replicas never prepare different requests for the same view and sequence number; 3A(ii) is also satisfied since a request that prepares at a non-faulty replica pre-prepares at at least  $f + 1$  non-faulty replicas. Condition 3A(iii) may not be satisfied initially, but the primary will eventually receive the request in a response to its status messages as discussed in Section 8. The primary uses this mechanism to request retransmission of missing requests as well as missing view-change and view-change acknowledgment messages. If a missing request arrives, this will trigger the decision procedure.

The decision procedure is efficient; our current implementation requires  $O(cK \times n^2)$  local steps at the primary

in the worst case (where  $c$  and  $K$  are the constants defined in Section 5.4). The normal case is even faster because most replicas propose identical values.

When the primary has selected a request for each slot, it broadcasts a new-view message to the other replicas. The new-view message has the form  $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{X} \rangle_{\alpha_p}$ . Here,  $\mathcal{V}$  contains a pair for each entry in  $\mathcal{S}$  consisting of the identifier of the sending replica and the digest of its view-change message, and  $\mathcal{X}$  identifies the checkpoint and request values picked. Then the primary updates its state to reflect the information in this message. It records all requests in  $\mathcal{X}$  as pre-prepared in view  $v + 1$  in its log. If it does not have the checkpoint with sequence number  $CP$  it also initiates the protocol to fetch the missing state. In any case the primary does not accept any prepare or commit messages with sequence number less than or equal to  $CP$  and does not send any pre-prepare message with such a sequence number.

The backups for view  $v + 1$  collect messages until they have a correct new-view message and a correct matching view-change message for each pair in  $\mathcal{V}$ . A backup may be unable to authenticate a view-change message for some replica with a pair in  $\mathcal{V}$ . This problem can occur if that replica is faulty because we use symmetric cryptography for message authentication and, therefore,  $p$  is unable to verify if the authenticator in a view-change message contains correct message authentication codes for the backups. The use of view-change-ack messages solves this problem. The primary only includes a pair for a view-change message in  $\mathcal{S}$  after it collects  $2f - 1$  matching view-change-ack messages from other replicas. This ensures that at least  $f + 1$  non-faulty replicas can vouch for the authenticity of every view-change message whose digest is in  $\mathcal{V}$ . Therefore, if the original sender of a view-change is uncooperative the primary retransmits that sender's view-change message and the non-faulty backups retransmit their view-change-acks. A backup can accept a view-change message whose authenticator is incorrect if it receives  $f$  view-change-acks that match the digest and identifier in  $\mathcal{V}$ . The retransmission of view-change, view-change-ack, and new-view messages is triggered by sending status messages.

After obtaining the new-view message and the matching view-change messages, the backups check if these messages support the decisions reported by the primary, by carrying out the decision procedure in Figure 3. If they do not, the replicas move immediately to view  $v + 2$ . Otherwise, they modify their state to account for the new information in a way similar to the primary. The only difference is that they multicast a prepare message for  $v + 1$  for each request they mark as pre-prepared and they may not have some of the requests whose digests were chosen by the primary in their log. These missing requests can be obtained using the status mechanism in Section 8.

Thereafter, the protocol proceeds as described in Section 5.3. Replicas redo the protocol for messages between  $n_m$  and  $n_M$  but they avoid re-executing client requests (by using their stored information about the last reply sent to each client).

If some replica changes its keys in the middle of a view change, it discards all the view-change protocol messages it already received with the old keys. Then it causes other replicas to re-send these messages using the status mechanism. If the primary has already sent the new-view message using the old key, this message will also be re-sent. The only difficulty occurs when some other replica is faulty and refuses to re-send its view-change message, and that message was used by the primary to make its decision. In this case the replicas will go on to another view change. If the primary changes its key there is no problem. If this occurs after it sent the new-view message there is nothing special to do; if it has not yet made its decision, it discards all its information about the new view and the replicas must re-send both their view-change and view-change-ack messages.

## 8 Obtaining Missing Information

This section describes the mechanisms for message retransmission and state transfer. The state transfer mechanism is necessary to bring replicas up to date when some of the messages they are missing were garbage collected.

### 8.1 Message Retransmission

We use a receiver-based recovery mechanism similar to SRM [12]: a replica  $i$  multicasts *status* messages that summarize its state; when other replicas receive a status message they retransmit messages they have sent in the past that  $i$  is missing. Status messages are sent periodically and when the replica detects that it is missing information (i.e., they also function as negative acks).

A replica  $i$  whose current view  $v$  is active multicasts messages with the format  $\langle \text{STATUS-ACTIVE}, v, ls, le, i, P, C \rangle_{\alpha_i}$ . Here,  $ls$  is the sequence number of the last stable checkpoint,  $le$  is the sequence number of the last request  $i$  has executed,  $P$  contains a bit for every sequence number between  $le$  and  $H$  (the high water mark) indicating whether that request prepared at  $i$ , and  $C$  is similar but indicates whether the request committed at  $i$ .

If the replica's current view is pending, it multicasts a status message with a different format to trigger retransmission of view-change protocol messages. The new format is  $\langle \text{STATUS-PENDING}, v, ls, le, i, n, V, R \rangle_{\alpha_i}$ . Here, the components with the same name have the same meaning,  $n$  is a flag that indicates whether  $i$  has the new-view message,  $V$  is a set with a bit for each replica that indicates whether  $i$  has accepted a view-change message for  $v$  from that replica, and  $R$  is a set with tuples  $\langle n, u \rangle$  indicating that  $i$  is missing a request that prepared in view  $u$  with sequence number  $n$ .  $R$  is used only if  $i$  is  $v$ 's primary to obtain missing requests to propagate to the new view.

If a replica  $j$  is unable to validate a status message, it sends its last new-key message to  $i$ . Otherwise,  $j$  sends messages it sent in the past that  $i$  may be missing. For example, if  $i$  is in a view less than  $j$ 's,  $j$  sends  $i$  its latest view-change message, or if  $j$  sent a commit for a sequence number with an unset bit in  $C$ , it retransmits that commit to  $i$ . In all these cases,  $j$  authenticates the messages it retransmits with the latest keys it received in a new-key message from  $i$ . This is important to ensure liveness with frequent key changes.

## 8.2 State Transfer

A replica may learn about a stable checkpoint beyond the high water mark in its log by receiving checkpoint messages or as the result of a view change. In this case, it uses the state transfer mechanism to fetch modifications to the service state that it is missing.

It is important for the state transfer mechanism to be efficient because it is used to bring a replica up to date during recovery and we perform proactive recoveries frequently. The key issues to achieving efficiency are reducing the amount of information transferred and reducing the burden imposed on other replicas. This mechanism must also ensure that the transferred state is correct. We start by describing our data structures and then explain how they are used by the state transfer mechanism.

### 8.2.1 Data Structures

We use of hierarchical state partitions to reduce the amount of information transferred. The root partition corresponds to the entire service state and each non-leaf partition is divided into  $s$  equal-sized, contiguous sub-partitions. We call the leaf partitions *pages*. For example, the experiments described in Section 10 were run with a hierarchy with three levels,  $s$  equal to 128, and 2KB-pages.

Each replica maintains one logical copy of the partition tree for each checkpoint. The copy is created when the checkpoint is taken and it is discarded when a later checkpoint becomes stable. The tree for a checkpoint stores a tuple  $\langle lm, d \rangle$  for each non-leaf partition and a tuple  $\langle lm, d, p \rangle$  for each page. Here,  $lm$  is the sequence number of the checkpoint at the end of the last checkpoint interval where the partition was modified,  $d$  is the digest of the partition, and  $p$  is the value of the page.

The digests are computed efficiently as follows. For a page,  $d$  is obtained by applying the MD5 hash function [34] to the string obtained by concatenating the index of the page within the state, its value of  $lm$  and  $p$ . For non-leaf partitions,  $d$  is obtained by applying MD5 to the string obtained by concatenating the index of the partition within its level, its value of  $lm$ , and the sum modulo a large integer of the digests of its sub-partitions. Thus, we apply AdHash [2] at each non-leaf level. This construction has the advantage that the digests for a checkpoint can be obtained efficiently by updating the digests from the previous checkpoint incrementally.

The copies of the partition tree are logical because we use copy-on-write so that only copies of the tuples modified since the checkpoint was taken are stored. This reduces the space and time overheads for maintaining these trees significantly. It is an extension of the technique used in [8].

## 8.2.2 Fetching State

The strategy to fetch state is to recurse down the hierarchy to determine which partitions are out of date. This reduces the amount of information about (both non-leaf and leaf) partitions that needs to be fetched.

A replica  $i$  multicasts  $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$  to all replicas to obtain information for the partition with index  $x$  in level  $l$  of the tree. Here,  $lc$  is the sequence number of the last checkpoint  $i$  knows for the partition, and  $c$  is either -1 or it specifies that  $i$  is seeking the value of the partition at sequence number  $c$  from replica  $k$ .

When a replica  $i$  determines that it needs to initiate a state transfer, it multicasts a fetch message for the root partition with  $lc$  equal to its last checkpoint. The value of  $c$  is non-zero when  $i$  knows the correct digest of the partition information at checkpoint  $c$ , e.g., after a view change completes  $i$  knows the digest of the checkpoint that propagated to the new view but might not have it.  $i$  also creates a new (logical) copy of the tree to store the state it fetches and initializes a table  $\mathcal{LC}$  in which it stores the number of the latest checkpoint reflected in the state of each partition in the new tree. Initially each entry in the table will contain  $lc$ .

If the designated replier,  $k$ , receives  $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$  and has a checkpoint for sequence number  $c$ , it sends back  $\langle \text{META-DATA}, c, l, x, P, k \rangle$ , where  $P$  is a set with a tuple  $\langle x', lm, d \rangle$  for each sub-partition of  $(l, x)$  with index  $x'$ , digest  $d$ , and  $lm > lc$ . Since  $i$  knows the correct digest for the partition value at checkpoint  $c$ , it can verify the correctness of the reply without the need for voting or even authentication. This reduces the burden imposed on other replicas.

The other replicas only reply to the fetch message if they have a stable checkpoint greater than  $lc$  and  $c$ . Their replies are similar to  $k$ 's except that  $c$  is replaced by the sequence number of their stable checkpoint and the message contains a MAC. These replies are necessary to guarantee progress when replicas have discarded a specific checkpoint requested by  $i$ .

Replica  $i$  retransmits the fetch message (choosing a different  $k$  each time) until it receives a valid reply from some  $k$  or  $f + 1$  equally fresh responses with the same sub-partition values for the same sequence number  $cp$  (greater than  $lc$  and  $c$ ). Then, it compares its digests for each sub-partition of  $(l, x)$  with those in the fetched information; it multicasts a fetch message for sub-partitions where there is a difference, and sets the value in  $\mathcal{LC}$  to  $c$  (or  $cp$ ) for the sub-partitions that are up to date. Since  $i$  learns the correct digest of each sub-partition at checkpoint  $c$  (or  $cp$ ) it can use the optimized protocol to fetch them.

The protocol recurses down the tree until  $i$  sends fetch messages for out-of-date pages. Pages are fetched like other partitions except that meta-data replies contain the digest and last modification sequence number for the page rather than sub-partitions, and the designated replier sends back  $\langle \text{DATA}, x, p \rangle$ . Here,  $x$  is the page index and  $p$  is the page value. The protocol imposes little overhead on other replicas; only one replica replies with the full page and it does not even need to compute a MAC for the message since  $i$  can verify the reply using the digest it already knows.

When  $i$  obtains the new value for a page, it updates the state of the page, its digest, the value of the last modification sequence number, and the value corresponding to the page in  $\mathcal{LC}$ . Then, the protocol goes up to its parent and fetches another missing sibling. After fetching all the siblings, it checks if the parent partition is *consistent*. A partition is consistent up to sequence number  $c$ , if  $c$  is the minimum of all the sequence numbers in  $\mathcal{LC}$  for its sub-partitions, and  $c$  is greater than or equal to the maximum of the last modification sequence numbers in its sub-partitions. If the parent partition is not consistent, the protocol sends another fetch for the partition. Otherwise, the protocol goes up again to its parent and fetches missing siblings.

The protocol ends when it visits the root partition and determines that it is consistent for some sequence number  $c$ . Then the replica can start processing requests with sequence numbers greater than  $c$ .

Since state transfer happens concurrently with request execution at other replicas, it may take some time for a replica to complete the protocol, e.g., each time it fetches a missing partition, it receives information about yet a later modification. This is unlikely to be a problem in practice (this intuition is confirmed by our experimental results). Furthermore, if the replica fetching the state ever is actually needed (because others have failed), the system will wait for it to catch up.

## 9 Implementation

We implemented our algorithm as a replication library. Some components of the library run on clients and others at the replicas. The interface we provide is very simple; it is shown in Figure 4.

Client:

```
int Byz_init_client(char *conf);
int Byz_invoke(Byz_req *req, Byz_rep *rep, bool read_only);
```

Server:

```
int Byz_init_replica(char *conf, char *mem, int size, Ucall execute);
void Byz_modify(char *mod, int size);
```

Server upcall:

```
int execute(Byz_req *req, Byz_rep *rep, int client);
```

Figure 4: The replication library API.

On the client side, the library provides a procedure to initialize the client using a configuration file, which contains the public keys and IP addresses of the replicas. The library also provides a procedure, *invoke*, that is called to cause an operation to be executed. This procedure carries out the client side of the protocol and returns the result when enough replicas have responded.

On the server side, we provide an initialization procedure that takes as arguments: a configuration file with the public keys and IP addresses of replicas and clients; the region of memory where the application state is stored, and a procedure to execute requests. When our system needs to execute an operation, it does an upcall to the *execute* procedure. This procedure carries out the operation as specified for the application, using the application state. As the application performs the operation, each time it is about to modify the application state, it calls the *modify* procedure to inform us of the locations about to be modified. This call allows us to maintain checkpoints and compute digests efficiently as described in Section 7.

We used the replication library to implement BFS, a Byzantine-fault-tolerant NFS [35] service. Figure 5 shows the architecture of BFS. Our implementation makes use of the optimizations described in [8]. These optimizations reduce the delays associated with read-only requests and the amount of state sent to the client for operations that return large results. More details about BFS can be found in [8].

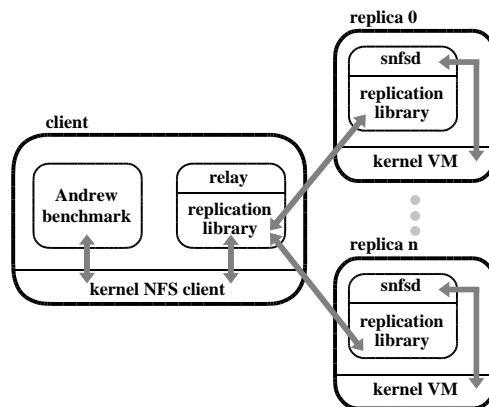


Figure 5: Replicated file system architecture

## 10 Performance Evaluation

This section uses a modified version of the Andrew benchmark [19] to evaluate the performance of BFS with proactive recoveries. It includes experiments to evaluate the performance degradation as the interval,  $T_k$ , between key changes and the interval,  $T_w$ , between recoveries decrease, and to measure the time to complete recovery in the normal case.

We also compare BFS and the NFS V2 implementation in Digital Unix. The performance of the commercial NFS implementation is used as a metric of acceptable performance: it is used daily by many users. This comparison shows that a secure configuration of BFS, with  $T_k$  equal to 30 seconds and  $T_w$  equal to 277 seconds, is only 8% slower than the Digital Unix NFS when running the modified Andrew benchmark.

*The results in this Section are preliminary; they were obtained with an older, less efficient version of the algorithms and they focus only on latency on an unloaded system.*

### 10.1 Experimental Setup

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 Seagate ST-32171N disk, with a peak transfer rate of 15.2 MB/s, an average read seek time of 9.4 ms, and an average rotational latency of 4.17 ms. 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,  $K$ , was 128 requests, which causes garbage collection to occur several times in any of the experiments. The maximum number of requests in the log was 256 (i.e., the constant  $c$  in the equation that determines the high water mark was 2).

### 10.2 Andrew Benchmark

The Andrew benchmark [19] 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 increased the size of the Andrew benchmark by a factor of 10: phase 1 and 2 create 10 copies of the source tree, and the other phases operate in all these copies. This modified Andrew benchmark produces 15.7 MB of data.

We use the Andrew benchmark to compare BFS with NFS-std, which is the NFS V2 implementation in Digital Unix. 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.

phase	BFS-base	NFS-std
1	1.1 (-68%)	3.4
2	79.3 (-13%)	91.3
3	58.0 (35%)	43.1
4	77.9 (41%)	55.38
5	391.1 (-2%)	398.0
total	607.5 (3%)	591.2

Table 1: Andrew benchmark: BFS-base vs NFS-std. The times are in seconds.

Table 1 shows the results for BFS-base vs NFS-std. BFS-base is BFS without recoveries and key changes. BFS-base takes only 3% more time to run the complete benchmark. The overhead of BFS relative to NFS-std is not the same for all phases. BFS-base is 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 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 [24]). NFS-std is faster than BFS-base in phases 3 and 4 because the client issues no synchronous operations during these

### 10.3 Key Lifetime

To evaluate the performance degradation as the key lifetime,  $T_k$ , decreases, we compared BFS-base with two other configurations: BFS-30 and BFS-60. In BFS-30,  $T_k$  is equal to 30 seconds (i.e., the client and each replica send a new-key message every 30 seconds) and  $T_k$  is equal to 60 seconds in BFS-60. Table 2 presents the result of this experiment.

phase	BFS/30	BFS/60	BFS-base
1	1.1 (0%)	1.1(0%)	1.1
2	84.3 (6%)	82.4 (4%)	79.3
3	59.0 (2%)	58.6 (1%)	58.0
4	81.3 (4%)	79.7 (2%)	77.9
5	407.3 (4%)	399.7 (2%)	391.1
total	633.0 (4%)	621.5 (2%)	607.5

Table 2: Andrew benchmark: BFS with different key lifetimes. The times are in seconds.

These results show that BFS can be configured with a very small key lifetime to improve security with a minimal performance degradation: BFS/30 and BFS/60 are only 4% and 2% slower than BFS-base. The main cause for this degradation is the cost of public-key cryptography. We use the Rabin-Williams public-key cryptosystem implementation from SFS [27] with a 1024-bit modulus. The costs for the cryptographic operations are listed in Table 3.

operation	time (ms)
sign	142.9
verify	0.7
encrypt	9.9
decrypt	132.3

Table 3: Public-Key Cryptography: Operation costs for 16 byte messages.

It takes 175.2 ms to generate a new-key message and 133.0 ms to process it at the receiver. Therefore, if each client and each replica sends a new-key message every 30 seconds each replica spends 2.4% of its time processing new-key messages. The remaining overhead is caused by the need to discard messages with stale keys, re-authenticate them with the new keys, and retransmit them.

It is interesting to predict the cost of running the Andrew benchmark using public-key signatures. There are approximately 24000 operations in the benchmark and each operation would have 5 signatures in the critical path of its execution. Even ignoring communication costs, this would result in an elapsed time of 17148 seconds. Our optimization of using authenticators improves performance by a factor of 28.

### 10.4 Recovery Time

The recovery time includes time to check that the code is correct and to reboot; to send a new-key message and a recovery request; to check that the service state is correct; and to fetch missing state and messages. If the replica is faulty it may also include time to fetch corrupted code or state.

Checking the correctness of the code will not take very long. It takes only 3.1 seconds to compute an MD5 digest of the kernel code (7.2 MB), and 1.83 seconds to compute the digest of the replica code (2 MB) starting from a cold file system cache. Furthermore, there is no need to check the code if it is kept in a read-only medium,

e.g., a write protected disk. Many modern disks have write protection that can only be disabled by physically changing a jumper setting [37]. This also obviates the need to fetch corrupted code.

We also measured the time to perform a reboot in our configuration; it takes 87 seconds to reboot the machine using the `reboot` command. When rebooting, the machine shuts down and restarts several servers that would not exist in a more secure configuration. We expect that rebooting could cost as little as 30 seconds in a suitably configured replica.

The remaining components of the recovery time for BFS were measured using the Andrew benchmark. We configured the replicas and client to refresh keys every 30 seconds. One of the replicas was configured to execute recoveries in succession with 30 seconds between the end of one recovery and the start of the next. To simulate the time to check the code and reboot, this replica slept for  $B$  seconds at the beginning of each recovery. Table 4 presents results for BFS/30/30, with  $B$  equal to 30, and BFS/30/90, with  $B$  equal to 90. Since the processes of checking the state for correctness and fetching missing pages are executed in parallel, Table 4 presents a single line *fetch and check* that accounts for the execution of the two processes.

	BFS/30/30	BFS/30/90
reboot	30.00	90.00
send new key	0.18	0.18
send request	0.14	0.14
fetch and check	8.95	10.52
total	39.27	100.84

Table 4: Recovery Time in seconds.

These results show that the recovery time is dominated by the time to reboot, and the time to send the new-key message and the recovery request is negligible. It also shows that the recovery time is very small. The time to fetch and check the state is lower in BFS/30/30 because it fetches fewer data pages and less meta-data information, as shown in Table 5. This happens because in BFS/30/30 the replica sleeps for less time when it simulates a reboot and, therefore, misses fewer modifications than in BFS/30/90.

Another, interesting observation is that our hierarchical scheme is very efficient at reducing the amount of data and meta-data fetched, e.g., only between 16% and 21% of the meta-data pages need to be fetched during recovery and in some recoveries only 2 data pages and 6 meta-data pages are fetched.

	BFS/30/30	BFS/30/90
#checked	15902	15280
#data fetched	577	1258
#meta-data fetched	21	27

Table 5: Recovery: Number of data pages checked and fetched, and meta-data pages fetched.

The file system size in this experiment was only 32 MB. This is a reasonable size for the state of some services, e.g., a certification authority, but most services will have more state. As the size of the service state increases, the first three components of the recovery time remain constant. Also, our hierarchical scheme only fetches blocks that are out of date and therefore the amount of missing state is proportional to the number of blocks affected by recent requests; this number is application dependent, but it is not dependent on the size of the application state. Therefore, the time to check the service state will become the dominant component of the recovery time. Using a new fast hash function called Tiger [1] we can digest 9.4 MB/s on the DEC Alpha 3000/400 and 55.3 MB/s on a DEC Alpha Workstation 500au. This means that for a state of 1 GB it will take about 109 seconds to check the correctness of the state (assuming the disk has sufficient bandwidth). On the new Alpha, the disk will be the bottleneck but, assuming one can read data at 15 MB/s, it will take about 68 seconds to check the correctness of the state. This allows us to use a period between recoveries on the order of tens of minutes for a state of a few gigabytes. We can reduce this by keeping most of the state in a write-protected disk and using copy-on-write to create writable copies of modified pages. If administrators transfer the modified pages back to the write-protected disk daily, the amount of state that needs to be checked is likely to be low.

## 10.5 Recovery Period

We also evaluated the impact of recovery on the performance of the client in the experimental setup described in the previous section. This setup corresponds to a recovery period of 277 seconds for BFS/30/30 and 523 seconds for BFS/30/90. Table 6 shows the results. The results show that even with very frequent recoveries the overhead of BFS relative to the standard NFS is low; it is below 12%. The overhead is higher for BFS/30/90 than for BFS/30/30 because BFS/30/90 fetches more information across the network.

system	time	overhead
BFS/30/90	661.6	11.9%
BFS/30/30	640.0	8.3%
BFS-base	607.5	2.7%
NFS-std	591.2	0%

Table 6: Andrew benchmark: BFS with recovery. Times in seconds.

When the service state is bigger, the time to complete recovery will become longer since it will take longer to check the service state, and so we will have to increase the recovery period. However, having a longer recovery should not have a significant impact on client performance since the other replicas can continue to process client requests. The recovering replica interferes with this proportional to how much state it needs to fetch, but as discussed in the preceding section, this is dependent on the amount of time it takes to check the code and reboot the system, and not on the amount of service state. This observation is supported by our experiment: the reason BFS/30/90 has more overhead than BFS/30/30 is because the replica gets further behind while it is rebooting and needs to fetch more state to catch up.

## 11 Related Work

Most previous work on replication techniques ignored Byzantine faults or assumed a synchronous system model, e.g., [22, 28, 23, 36, 9, 14]. But a practical replication system must work correctly in asynchronous systems. Recovery has been studied before in asynchronous systems that tolerate benign faults, e.g., Harp [24], but not in asynchronous Byzantine-fault tolerant systems. Therefore, the best that these systems [33, 21, 26, 8] could guarantee was safety provided less than  $1/3$  of the replicas were faulty during the lifetime of the system. This guarantee is too weak for long-lived systems. Our system improves this guarantee by recovering replicas proactively; it can tolerate any number of faults if less than  $1/3$  of the replicas become faulty within a window of vulnerability (which can be made very small under normal load conditions with low impact on performance).

Rampart [32, 33] and SecureRing [21] provide group membership protocols. These protocols can be used to implement recovery in the presence of benign faults as follows: replicas that are suspected to be faulty are removed from the group and after they recover they are re-added to the group. But this does not work in the presence of Byzantine faults for two reasons. First, the system may be unable to provide safety if a replica that is not faulty is removed from the group to be recovered. Second, there is no mechanism to prevent attackers from impersonating recovered replicas that they controlled in the past.

The problem of efficient state transfer has been ignored by previous work on Byzantine-fault-tolerant replication. We present an efficient state transfer algorithm that enables frequent proactive recoveries with low performance degradation. The algorithm is also unusual because it is highly asynchronous. In replication algorithms for benign faults, e.g., [24, 5], replicas retain a checkpoint of the state and messages in their log until the recovering replica is brought up-to-date. This could open an avenue for a denial-of-service attack in the presence of Byzantine faults. Instead, in our algorithm, replicas are free to garbage collect information and are never delayed by the recovery.

The concept of a system that can tolerate more than  $f$  faults provided no more than  $f$  nodes in the system become faulty in some time window was introduced in [29]. This concept has previously been applied in synchronous systems to secret-sharing schemes [17], threshold cryptography [18], and more recently secure information storage and retrieval [13] (which provides single-writer single-reader replicated variables). But our algorithm is more general; it allows a group of nodes in an asynchronous system to implement an arbitrary state machine.

Our algorithm is most similar to the one presented in [8] but it provides recovery and a state transfer mechanism, which were not provided in [8]. Furthermore, it provides a new view change mechanism that enables both recovery and an important optimization — the use of message authentication codes instead of public-key cryptography, which was the major performance bottleneck in previous systems [31, 25].

The protocol we use during normal-case operation is similar to the Byzantine agreement algorithm in [6], which also does not use public-key cryptography, but the algorithm in [6] does not provide view changes and, therefore, does not provide liveness if the primary fails.

## 12 Conclusions

This paper has described a new state-machine replication system that offers both integrity and high availability in the presence of Byzantine faults. The new system can be used to implement real services because it performs well, works in asynchronous systems like the Internet, and recovers replicas to enable long-lived services.

The system is optimized for configurations with a moderate number of replicas (e.g., at most 31), which we expect will be sufficient for almost all applications; it is likely to perform poorly for configurations that are much larger. The system scales well with the number of clients because we only keep the last replies and session keys for active clients. We store the last request timestamp for all clients but this represents less than 16MB on disk for one million clients; only the timestamps for active clients are cached in main memory.

The system described here improves the security of previous systems by recovering replicas proactively and frequently. It can tolerate any number of faults provided less than 1/3 of the replicas become faulty within a window of vulnerability. This window can be made very small (e.g., a few minutes) under normal load conditions and when the attacker does not corrupt replicas' copies of the service state. Additionally, our system provides *intrusion detection*; it detects denial-of-service attacks aimed at increasing the window and the corruption of the state of a recovering replica. Therefore, a successful attack can escape detection only if the attacker compromises more than 1/3 of the replicas within a small time interval.

Recovery from Byzantine faults is harder than recovery from benign faults for several reasons: the recovery protocol itself needs to tolerate other Byzantine-faulty replicas; replicas must be recovered proactively; and attackers must be prevented from impersonating recovered replicas that they controlled. For example, the last requirement prevents signatures in messages from being valid indefinitely. However, this leads to a further problem, since replicas may be unable to prove to a third party that some message they received is authentic (because its signature is no longer valid). All previous state-machine replication algorithms relied on such proofs. Our algorithm does not rely on these proofs and has the added advantage of enabling the use of symmetric cryptography for authentication of all protocol messages. This eliminates the use of public-key cryptography, the major performance bottleneck in previous systems.

The algorithm has been implemented as a generic program library with a simple interface that can be used to provide Byzantine-fault-tolerant versions of different services. We used the library to implement BFS, a replicated NFS service, and ran experiments to determine the performance impact of our techniques by comparing BFS with an unreplicated NFS. The experiments show that it is possible to use our algorithm to implement real services with latency close to that of an unreplicated service — the latency of BFS on the Andrew benchmark [19] is only 3% worse than that of the standard NFS implementation in Digital Unix. Furthermore, they suggest that the window of vulnerability can be made very small ( $\approx 5$  minutes) with less than 9% degradation in performance relative to the standard NFS implementation.

## References

- [1] Ross Anderson and Eli Biham. Tiger: A fast new hash function. Technical report, Cambridge University England, 1995.
- [2] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology – Eurocrypt 97*, 1997.
- [3] M. Bellare and P. Rogaway. Optimal asymmetric encryption - How to encrypt with RSA. In *Advances in Cryptology - EUROCRYPT 94, Lecture Notes in Computer Science, Vol. 950*. Springer-Verlag, 1995.

- [4] M. Bellare and P. Rogaway. The exact security of digital signatures- How to sign with RSA and Rabin. In *Advances in Cryptology - EUROCRYPT 96, Lecture Notes in Computer Science, Vol. 1070*. Springer-Verlag, 1996.
- [5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems*, volume 9(3), August 1991.
- [6] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–240, 1985.
- [7] 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.
- [8] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (to appear)*, New Orleans, LA, February 1999.
- [9] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *15th International Conference on Fault Tolerant Computing, Ann Arbor, Mi.*, June 1985.
- [10] Dallas Semiconductor Corporation. DS1286 Watchdog Timekeeper. <http://www.dalsemi.com>, 1999.
- [11] Dallas Semiconductor Corporation. ibutton. <http://www.ibutton.com>, 1999.
- [12] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L.H.Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6), August 1995.
- [13] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure Distributed Storage and Retrieval. *To appear in Theoretical Computer Science*, 1999.
- [14] J. Garay and Y. Moses. Fully polynomial byzantine agreement for  $n > 3t$  processors in  $t+1$  rounds. *SIAM Journal of Computing*, 27(1):247–290, February 1998.
- [15] L. Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, January 1992.
- [16] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.
- [17] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In *Proc. of CRYPTO'95*, 1995.
- [18] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proc. of the 1997 ACM Conference on Computers and Communication Security*, 1997.
- [19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [20] International Business Machines Corporation. <http://www.ibm.com/>, 1999.
- [21] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. of the Hawaii International Conference on System Sciences*, Hawaii, January 1998.
- [22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.
- [23] L. Lamport. The Part-Time Parliament. Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.
- [24] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 226–238. ACM Press, 1991.
- [25] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. In *Proc. of the 9th Computer Security Foundations Workshop*, pages 9–17, Ireland, June 1996.
- [26] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.
- [27] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Kiawah Island, SC, December 1999.

- [28] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.
- [29] R. Ostrovsky and M. Yung. How to withstand mobile virus attack. In *Proc. of the 19th Symposium on Principles of Distributed Computing*, pages 51–59. ACM, October 1991.
- [30] R.Canetti, S.Halevi, and A.Herzberg. Maintaining authenticated communication in the presence of break-ins. In *Proc. of the 1997 ACM Conference on Computers and Communication Security*, 1997.
- [31] M. Reiter. Secure Agreement Protocols. In *Proc. of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.
- [32] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pages 99–110, 1995.
- [33] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.
- [34] R. Rivest. The MD5 message-digest algorithm. Internet RFC-1321, April 1992.
- [35] R. Sandberg et al. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.
- [36] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [37] Seagate Technology, Inc. <http://www.seagate.com/>, 1997.
- [38] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM Computer Communications Review*, 22(5):29–38, 1992.
- [39] H. C. Williams. A Modification of the RSA Public-Key Encryption Procedure. *IEEE Transactions on Information Theory*, 26(6), November 1980.