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

Abstract

This paper describes an asynchronous state-machine replication system that tolerates Byzantine faults, which can be caused by malicious attacks or software errors. Our system is the first to recover Byzantine-faulty replicas proactively and it performs well because it uses symmetric rather than public-key cryptography for authentication. The recovery mechanism allows us to tolerate any number of faults over the lifetime of the system provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability that is small under normal conditions. The window may increase under a denial-of-service attack but we can detect and respond to such attacks. The paper presents results of experiments showing that overall performance is good and that even a small window of vulnerability has little impact on service latency.

1 Introduction

This paper describes a new system for asynchronous state-machine replication [17, 28] that offers both integrity and high availability in the presence of Byzantine faults. Our system is interesting for two reasons: it improves security by recovering replicas proactively, and it is based on symmetric cryptography, which allows it to perform well so that it can be used in practice to implement real services.

Our system continues to function correctly even when some replicas are compromised by an attacker; this is worthwhile because the growing reliance on online information services makes malicious attacks more likely and their consequences more serious. The system also survives nondeterministic software bugs and software bugs due to aging (e.g., memory leaks). Our approach improves on the usual technique of rebooting the system because it refreshes state automatically, staggers recovery so that individual replicas are highly unlikely to fail simultaneously, and has little impact on overall system performance. Section 4.7 discusses the types of faults tolerated by the system in more detail.

Because of recovery, our system can tolerate any number of faults over the lifetime of the system, provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability. The best that could be guaranteed previously was correct behavior if fewer than $1/3$ of the replicas failed during the lifetime of a system. Our previous work [6] guaranteed this and other systems [26, 16] provided weaker guarantees. Limiting the number of failures that can occur in a finite window is a synchrony assumption but such an assumption is unavoidable: since Byzantine-faulty replicas can discard the service state, we must bound the number of failures that can occur before recovery completes. But we require no synchrony assumptions to match the guarantee provided by previous systems. We compare our approach with other work in Section 7.

The window of vulnerability can be small (e.g., a few minutes) under normal conditions. 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, integrity can be preserved even when there is a denial-of-service attack.

The paper describes a number of new techniques needed to solve the problems that arise when providing recovery from Byzantine faults:

Proactive recovery. A Byzantine-faulty replica may appear to behave properly even when broken; therefore recovery must be proactive to prevent an attacker from compromising the service by corrupting $1/3$ of the replicas without being detected. Our algorithm recovers replicas periodically independent of any failure detection mechanism. However a recovering replica may not be faulty and recovery must not cause it to become faulty, since otherwise the number of faulty replicas could 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.

Fresh messages. An attacker must be prevented from impersonating a replica that was faulty after it recovers. This can happen if the attacker learns the keys used to authenticate messages. Furthermore even if messages are signed using a secure cryptographic co-processor, an attacker might be able to authenticate bad messages while it controls a faulty replica; these messages could be replayed later to compromise safety. To solve this problem, we define a notion of authentication *freshness*

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

and replicas 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 [26, 16], including the one we described in [6], relied on such proofs. Our current algorithm does not, and this has the added advantage of enabling the use of symmetric cryptography for authentication of all protocol messages. This eliminates most use of public-key cryptography, the major performance bottleneck in previous systems.

Efficient state transfer. State transfer is harder in the presence of Byzantine faults and efficiency is crucial to enable frequent recovery with little impact on performance. To bring a recovering replica up to date, the state transfer mechanism checks 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 hash chaining and incremental cryptography [1]; the mechanism tolerates Byzantine-faults and state modifications while transfers are in progress.

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. The paper describes experiments that compare the performance of a replicated NFS implemented using the library with an unreplicated NFS. The results show that the performance of the replicated system without recovery is close to the performance 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 (2 to 10 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 states the properties provided by our algorithm; and Section 4 describes the algorithm. Our implementation is described in Section 5 and some performance experiments are presented in Section 6. Section 7 discusses related work. Our conclusions are presented in Section 8.

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 the SFS [21] implementation of a Rabin-Williams public-key cryptosystem with a 1024-bit modulus to establish 128-bit session keys. All messages are then authenticated using message authentication codes (MACs) [2] computed using these keys. Message digests are computed using MD5 [27].

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 forge signatures or MACs without knowing the corresponding keys, 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 [6, 26, 16] 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. To tolerate more faults we need additional assumptions: we must mutually authenticate a faulty replica that recovers to the other replicas, and we need a reliable mechanism to trigger periodic recoveries. These 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, or the security chip in the motherboard of the IBM PC 300PL. 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.

Read-Only 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. Most motherboards can be configured such that it is necessary to have physical access to the machine to modify the BIOS.

Watchdog Timer. Each replica has a *watchdog timer* that periodically interrupts processing and hands control to a *recovery monitor*, which is stored in the read-only 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.

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 system administrators to perform recovery.

Note that all previous proactive security algorithms [24, 13, 14, 3, 10] assume the entire program run by a replica is in read-only memory so that it cannot be modified by an attacker. Most also assume that there are authenticated channels between the replicas that continue to work even after a replica recovers from a compromise. These assumptions would be sufficient to implement our algorithm but they are less likely to hold in practice. We only require a small monitor in read-only memory and use the secure co-processors to establish new session keys between the replicas after a recovery.

The only work on proactive security that does not assume authenticated channels is [3], 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.

We also assume clients have a secure co-processor; this simplifies the key exchange protocol between clients and replicas but it could be avoided by adding an extra round to this protocol.

3 Algorithm Properties

Our algorithm is a form of *state machine* replication [17, 28]: 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; they can perform arbitrary computations.

The service is implemented by a set of replicas $\mathcal{R}$ and each replica is identified using an integer in $\{0, ..., |\mathcal{R}| - 1\}$. Each replica maintains a copy of 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 MACs).

Like all state machine replication techniques, we impose two requirements on replicas: they must start in the same state, and 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). We can handle some common forms of non-determinism using the technique we described in [6].

Our algorithm ensures safety for an execution provided at most f replicas become faulty within a window of vulnerability of size T_v. Safety means that the replicated service satisfies linearizability [12, 5]: it behaves like a centralized implementation that executes operations atomically one at a time. Our algorithm provides safety regardless of how many faulty clients are using the service (even if they collude with faulty replicas). We will discuss the window of vulnerability further in Section 4.7.

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) denial-of-service attacks do not last forever, i.e., 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.

4 Algorithm

The algorithm works as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. 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 $p = v \textit{ mod } |\mathcal{R}|$, 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 to select a new primary. Viewstamped Replication [23] and Paxos [18] use 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 some *statement* is correct and coming from different replicas. An example of a statement is: "the result of the operation requested by a client 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. 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 even when f replicas are faulty.

Our earlier algorithm [6] used the same basic ideas but it did not provide recovery. 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. We avoid this problem by introducing a notion of *freshness*; replicas reject messages that are not fresh. But this raises another problem: the view change protocol in [6] relied on the exchange of certificates between replicas and this may be impossible because some of the messages in a certificate may no longer be fresh. Section 4.5 describes a new view change protocol that solves this problem and also eliminates the need for expensive public-key cryptography.

To provide liveness with the new protocol, a replica must be able to fetch missing state that may be held by a single correct replica whose identity is not known. In this case, voting cannot be used to ensure correctness of the data being fetched and it is important to prevent a faulty replica from causing the transfer of unnecessary or corrupt data. Section 4.6 describes a mechanism to obtain missing messages and state that addresses these issues and that is efficient to enable frequent recoveries.

The sections below describe our algorithm. Sections 4.2 and 4.3, which explain normal-case request processing, are similar to what appeared in [6]. They are presented here for completeness and to highlight some subtle changes.

4.1 Message Authentication

We use MACs to authenticate all 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.

Some messages in the protocol contain a single MAC computed using UMAC32 [2]; we denote such a message as $\langle m \rangle_{\mu_{ij}}$, 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 $\langle \text{NEW-KEY}, i, ..., \{k_{j,i}\}_{\epsilon_j}, ..., t \rangle_{\sigma_i}$. 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 suppress-replay attacks [11].) 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 ensures that correct nodes only accept certificates with *equally fresh* messages, i.e., messages authenticated with keys created in the same refreshment phase.

4.2 Processing Requests

We use 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.

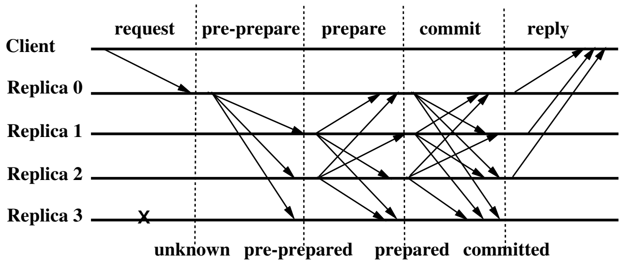

Figure 1: Normal Case Operation. Replica 0 is the primary, and replica 3 is faulty

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 4.3.

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 [6].

When the primary p receives a request m from a client, it assigns a sequence number n to m. Then it multicasts a pre-prepare message with the assignment to 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 if 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 4.3.

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 certificate with a pre-prepare message and $2f$ 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 $\langle \text{COMMIT}, v, n, d, i\rangle_{\alpha_i}$ saying it prepared the request. This starts the commit phase. When a replica has accepted a certificate with $2f + 1$ commit messages for the same sequence number n and digest d from different replicas (including itself), it marks the request 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 ensures information about committed requests is 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 safety.

After executing the requested operation, replicas send a reply to the client c. The reply has the form $\langle \text{REPLY}, v, t, c, i, r\rangle_{\mu_{ic}}$ where 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 v so that clients can track the current primary.

The client waits for a certificate with $f + 1$ 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 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.

4.3 Garbage Collection

Replicas can discard entries from their log once the corresponding requests 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 after 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., $K = 128$) is executed. We will refer to the states produced by the execution of these requests as *checkpoints*.

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. A replica maintains several logical copies of the service state: the current state and some previous checkpoints. Section 4.6 describes how we manage checkpoints efficiently.

Each replica waits until it has a certificate containing $2f+1$ valid checkpoint messages for sequence number n with the same digest d sent by different replicas (including possibly its own message). At this point, the checkpoint is said 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 added to the log). The low-water mark h is equal to the sequence number of the last stable checkpoint and the high water mark is $H = h + L$, where L is the log size. The log size is obtained by multiplying K by a small constant factor (e.g., 2) that is big enough so that replicas do not stall waiting for a checkpoint to become stable.

4.4 Recovery

The recovery protocol makes faulty replicas behave correctly again to allow the system to tolerate more than f faults over its lifetime. To achieve this, the protocol ensures that after a replica recovers it is running correct code; it cannot be impersonated by an attacker; and it has correct, up-to-date state.

Reboot. Recovery is proactive — it starts periodically when the watchdog timer goes off. The recovery monitor saves the replica's state (the log and the service state) to disk. Then it reboots the system with correct code and restarts the replica from the saved state. The correctness of the operating system and service code is ensured by storing them in a read-only medium (e.g., the Seagate Cheetah 18LP disk can be write protected by physically closing a jumper switch). Rebooting restores the operating system data structures and removes any Trojan horses.

After this point, the replica's code is correct and it did not lose its state. The replica must retain its state and use it to process requests even while it is recovering. This is vital to ensure both safety and liveness in the common case when the recovering replica is not faulty; otherwise, recovery could cause the $f + 1$st fault. But if the recovering replica was faulty, the state may be corrupt and the attacker may forge messages because it knows the MAC keys used to authenticate both incoming and outgoing messages. The rest of the recovery protocol solves these problems.

The recovering replica i starts by discarding the keys it shares with clients and it multicasts a new-key message to change the keys it uses to authenticate messages sent by the other replicas. This is important if i was faulty because otherwise the attacker could prevent a successful recovery by impersonating any client or replica.

Run estimation protocol. Next, i runs a simple protocol to estimate an upper bound, H_M, on the high-water mark that it would have in its log if it were not faulty. It discards any entries with greater sequence numbers to bound the sequence number of corrupt entries in the log.

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 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 and the last request prepared at j respectively. i keeps retransmitting the query message and processing replies; it keeps the minimum value of c and the maximum value of p it received from each replica. It also keeps its own values of c and p.

The recovering replica uses the responses to select H_M as follows: $H_M = L + c_M$ where L is the log size and c_M is a value c received from replica j such that $2f$ replicas other than j reported values for c less than or equal to c_M and f replicas other than j reported values of p greater than or equal to c_M.

For safety, c_M must be greater than any stable checkpoint so that i will not discard log entries when it is not faulty. This is insured because if a checkpoint is stable it will have been created by at least $f + 1$ non-faulty replicas and it will have a sequence number less than or equal to any value of c that they propose. The test against p ensures that c_M is close to a checkpoint at some non-faulty replica since at least one non-faulty replica reports a p not less than c_M; 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 at least $f + 1$ correct replicas.

After this point i participates in the protocol as if it were not recovering but it will not send any messages above H_M until it has a correct stable checkpoint with sequence number greater than or equal to H_M.

Send recovery request. Next i sends a recovery request with the form: $\langle \text{REQUEST}, \langle \text{RECOVERY}, H_M\rangle, t, i\rangle_{\sigma_i}$. This message is produced by the cryptographic co-processor and t is the co-processor's counter to prevent replays. The other replicas reject the request if it is a replay or if they accepted a recovery request from i recently (where recently can be defined as half of the watchdog period). This is important to prevent a denial-of-service attack where non-faulty replicas are kept busy executing recovery requests.

The recovery request is treated like any other request: it is assigned a sequence number n_R and it goes through the usual three phases. But when another replica executes the recovery request, it sends its own new-key message. Replicas also send a new-key message when they fetch missing state (see Section 4.6) and determine that it reflects the execution of a new recovery request. This is important because these keys are known to the attacker if the recovering replica was faulty. By changing these keys, we bound the sequence number of messages forged by the attacker that may be accepted by the other replicas — they are guaranteed not to accept forged messages with sequence numbers greater than the maximum high water mark in the log when the recovery request executes, i.e., $H_R = \lfloor n_R/K\rfloor \times K + L$.

The reply to the recovery request includes the sequence number n_R. Replica 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*, $H = max(H_M, H_R)$. It also computes a valid view (see Section 4.5); 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.

Check and fetch state. While i is recovering, it uses the state transfer mechanism discussed in Section 4.6 to determine what pages of the state are corrupt and to fetch pages that are out-of-date or corrupt.

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, the recovery request ensures that other replicas will not accept forged messages with sequence numbers greater than H.

Our protocol 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.

4.5 View Change Protocol

The view change protocol provides liveness by allowing the system to make progress when the current primary fails. The protocol must preserve safety: it must ensure that non-faulty replicas agree on the sequence numbers of committed requests across views. In addition, the protocol must provide liveness: it must ensure that non-faulty replicas stay in the same view long enough for the system to make progress, even in the face of a denial-of-service attack.

The new view change protocol uses the techniques described in [6] to address liveness but uses a different approach to preserve safety. Our earlier approach relied on certificates that were valid indefinitely. In the new protocol, however, the fact that messages can become stale means that a replica cannot prove the validity of a certificate to others. Instead the new protocol relies on the group of replicas to validate each statement that some replica claims has a certificate. The rest of this section describes the new protocol.

Data structures. Replicas record information about what happened in earlier views. This information is maintained in two sets, the *PSet* and the *QSet*. A replica also stores the requests corresponding to the entries in these sets. These sets only contain information for sequence numbers between the current low and high water marks in the log; therefore only limited storage is required. The sets allow the view change protocol to work properly even when more than one view change occurs before the system is able to continue normal operation; the sets are usually empty while the system is running normally.

The *PSet* at replica i stores information about requests that have prepared at i in previous views. 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 prepared at i in a later view.

The *QSet* stores information about requests that have pre-prepared at i in previous views (i.e., requests for which i has sent a pre-prepare or prepare message). Its entries are tuples $\langle n, \{..., \langle d_k, v_k\rangle, ...\}\rangle$ meaning that for each k, v_k is the latest view in which a request pre-prepared with sequence number n and digest d_k at i.

View-change messages. View changes are triggered when the current primary is suspected to be faulty (e.g., when a request from a client is not executed after some period of time; see [6] for details). When a backup i suspects the primary for view v is faulty, it enters view $v+1$ and 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 the *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. The computation bounds the size of each tuple in *QSet*; it retains only pairs corresponding to $f+2$ distinct requests (corresponding to possibly f messages from faulty replicas, one message from a good replica, and one special null message as explained below). Therefore the amount of storage used is bounded.

View-change-ack messages. Replicas collect view-change messages for $v+1$ and send acknowledgments for them to $v+1$'s primary, p. The acknowledgments have the form $\langle \text{VIEW-CHANGE-ACK}, v+1, i, j, d\rangle_{\mu_{ip}}$ where i is the identifier of the sender, d is the digest of the view-change message being acknowledged, and j is the replica that sent that view-change message. These acknowledgments allow the primary to prove authenticity of view-change messages sent by faulty replicas as explained later.

let v be the view before the view change, L be the size of the log, and h be the log's low water mark

for all n such that $h < n \leq h + L$ do
  if request number n with digest d is prepared or committed in view v then
    add $\langle n, d, v\rangle$ to $\mathcal{P}$
  else if $\exists \langle n, d', v'\rangle \in$ *PSet* then
    add $\langle n, d', v'\rangle$ to $\mathcal{P}$
  if request number n with digest d is pre-prepared, prepared or committed in view v then
    if $\neg\exists \langle n, D\rangle \in$ *QSet* then
      add $\langle n, \{\langle d, v\rangle\}\rangle$ to $\mathcal{Q}$
    else if $\exists \langle d, v'\rangle \in D$ then
      add $\langle n, D \cup \{\langle d, v\rangle\} - \{\langle d, v'\rangle\}\rangle$ to $\mathcal{Q}$
    else if $|D| > f+1$ then
      remove entry with lowest view number from D
      add $\langle n, D \cup \{\langle d, v\rangle\}\rangle$ to $\mathcal{Q}$
  else if $\exists \langle n, D\rangle \in$ *QSet* then
    add $\langle n, D\rangle$ to $\mathcal{Q}$

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

New-view message construction. 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 $\mathcal{S}$ after receiving $2f-1$ view-change-acks for i's view-change message from other replicas. Each entry in $\mathcal{S}$ is for a different replica.

let $D = \{\langle n, d\rangle \mid \exists\ 2f+1$ messages $m \in \mathcal{S} : m.ls \leq n$
    $\wedge\ \exists\ f+1$ messages $m \in \mathcal{S} : \langle n, d\rangle \in m.\mathcal{C}\}$
if $\exists \langle h, d\rangle \in D : \forall \langle n', d'\rangle \in D : n' \leq h$ then
  select checkpoint with digest d and number h
else exit

for all n such that $h < n \leq h + L$ do
  A. if $\exists\, m \in \mathcal{S}$ with $\langle n, d, v\rangle \in m.\mathcal{P}$ that verifies:
    A1. $\exists\ 2f+1$ messages $m' \in \mathcal{S}$:
      $m'.ls < n \wedge m'.\mathcal{P}$ has no entry for n **or**
      $\exists \langle n, d', v'\rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$
    A2. $\exists\ f+1$ messages $m' \in \mathcal{S}$:
      $\exists \langle n, \{..., \langle d', v'\rangle, ...\}\rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$
    A3. the primary has the request with digest d
  then select the request with digest d for number n
  B. else if $\exists\ 2f+1$ messages $m \in \mathcal{S}$ such that
      $m.ls < n \wedge m.\mathcal{P}$ has no entry for n
  then select the null request for number n

Figure 3: Decision procedure at the primary.

The new primary uses the information in $\mathcal{S}$ and the decision procedure sketched in Figure 3 to choose a checkpoint and a set of requests. This procedure runs each time the primary receives new information, e.g., when it adds a new message to $\mathcal{S}$.

The primary starts by selecting the checkpoint that is going to be the starting state for request processing in the new view. It picks the checkpoint with the highest number h from the set of checkpoints that are known to be correct and that have numbers higher than the low water mark in the log of at least $f+1$ non-faulty replicas. The last condition is necessary for safety; it ensures that the ordering information for requests that committed with numbers higher than h is still available.

Next, the primary selects a request to pre-prepare in the new view for each sequence number between h and $h+L$ (where L is the size of the log). For each number n that was assigned to some request m that committed in a previous view, the decision procedure selects m to pre-prepare in the new view with the same number. This ensures safety because no distinct request can commit with that number in the new view. For other numbers, the primary may pre-prepare a request that was in progress but had not yet committed, or it might select a special *null* request that goes through the protocol as a regular request but whose execution is a no-op.

We now argue informally that this procedure will select the correct value for each sequence number. If a request m committed at some non-faulty replica with number n, it prepared at at least $f+1$ non-faulty replicas and the view-change messages sent by those replicas will indicate that m prepared with number n. Any set of at least $2f+1$ view-change messages for the new view must include a message from one of the non-faulty replicas that prepared m. Therefore, the primary for the new view will be unable to select a different request for number n because no other request will be able to satisfy conditions A1 or B (in Figure 3).

The primary will also be able to make the right decision eventually: condition A1 will be satisfied because there are $2f+1$ non-faulty replicas and non-faulty replicas never prepare different requests for the same view and sequence number; A2 is also satisfied since a request that prepares at a non-faulty replica pre-prepares at at least $f+1$ non-faulty replicas. Condition A3 may not be satisfied initially, but the primary will eventually receive the request in a response to its status messages (discussed in Section 4.6). When a missing request arrives, this will trigger the decision procedure to run.

The decision procedure ends when the primary has selected a request for each number. This takes $O(L \times |\mathcal{R}|^3)$ local steps in the worst case but the normal case is much faster because most replicas propose identical values. After deciding, the primary multicasts a new-view message to the other replicas with its decision. 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 selected.

New-view message processing. The primary updates its state to reflect the information in the new-view 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 h it also initiates the protocol to fetch the missing state (see Section 4.6.2). In any case the primary does not accept any prepare or commit messages with sequence number less than or equal to h 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 $\mathcal{V}$. If some replica changes its keys in the middle of a view change, it has to discard all the view-change protocol messages it already received with the old keys. The message retransmission mechanism causes the other replicas to re-send these messages using the new keys.

If a backup did not receive one of the view-change messages for some replica with a pair in $\mathcal{V}$, the primary alone may be unable to prove that the message it received is authentic because it is not signed. 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 $\mathcal{V}$.

After obtaining the new-view message and the matching view-change messages, the backups check whether 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. Thereafter, the protocol proceeds as described in Section 4.2.

The replicas use the status mechanism in Section 4.6 to request retransmission of missing requests as well as missing view-change, view-change acknowledgment, and new-view messages.

4.6 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.

4.6.1 Message Retransmission

We use a receiver-based recovery mechanism similar to SRM [8]: a replica i multicasts small *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).

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. In all cases, j authenticates 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.

Clients retransmit requests to replicas until they receive enough replies. They measure response times to compute the retransmission timeout and use a randomized exponential backoff if they fail to receive a reply within the computed timeout.

4.6.2 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 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.

Data Structures. We use 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 leaf partitions *pages* and interior partitions meta-data. For example, the experiments described in Section 6 were run with a hierarchy with four levels, s equal to 256, and 4KB 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 meta-data 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 [27] to the string obtained by concatenating the index of the page within the state, its value of lm and p. For meta-data, 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 [1] at each meta-data 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 checkpoints significantly.

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 $\langle \text{FETCH}, l, x, lc, c, k, i\rangle_{\alpha_i}$ is received by the designated replier, k, and it 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 and other replicas are free to garbage collect checkpoints, 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.

4.7 Discussion

Our system ensures safety and liveness for an execution τ 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 τ and unknown to the algorithm. T_k is the maximum key refreshment period in τ for a non-faulty node, and T_r is the maximum time between when a replica fails and when it recovers from that fault in τ.

The message authentication mechanism from Section 4.1 ensures non-faulty nodes only accept certificates with messages generated within an interval of size at most $2T_k$.[1] The bound on the number of faults within T_v ensures there are never more than f faulty replicas within any interval of size at most $2T_k$. Therefore, safety and liveness are provided because non-faulty nodes never accept certificates with more than f bad messages.

We have little control over the value of T_v because T_r may be increased by a denial-of-service attack, but we have good control over T_k and the maximum time between watchdog timeouts, 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. Section 6 analyzes this tradeoff.

The value of T_w should be set based on R_n, the time it takes to recover a non-faulty replica under normal load conditions. 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 (i.e., no attack).

Another issue is the bound f on the number of faults. Our replication technique is not useful if there is a strong positive correlation between the failure probabilities of the replicas; the probability of exceeding the bound may not be lower than the probability of a single fault in this case. Therefore, it is important to take steps to increase diversity. One possibility is to have diversity in the execution environment: the replicas can be administered by different people; they can be in different geographic locations; and they can have different configurations (e.g., run different combinations of services, or run schedulers with different parameters). This improves resilience to several types of faults, for example, attacks involving physical access to the replicas, administrator attacks or mistakes, attacks that exploit weaknesses in other services, and software bugs due to race conditions. Another possibility is to have software diversity; replicas can run different operating systems and different implementations of the service code. There are several independent implementations available for operating systems and important services (e.g. file systems, data bases, and WWW servers). This improves resilience to software bugs and attacks that exploit software bugs.

Even without taking any steps to increase diversity, our proactive recovery technique increases resilience to nondeterministic software bugs, to software bugs due to aging (e.g., memory leaks), and to attacks that take more time than T_v to succeed. It is possible to improve security further by exploiting software diversity across recoveries. One possibility is to restrict the service interface at a replica after its state is found to be corrupt. Another potential approach is to use obfuscation and randomization techniques [7, 9] to produce a new version of the software each time a replica is recovered. These techniques are not very resilient to attacks but they can be very effective when combined with proactive recovery because the attacker has a bounded time to break them.

5 Implementation

We implemented the algorithm as a library with a very simple interface (see Figure 4). Some components of the library run on clients and others at the replicas.

On the client side, the library provides a procedure

[1] It would be T_k except that during view changes replicas may accept messages that are claimed authentic by $f+1$ replicas without directly checking their authentication token.

```
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, UC exec);
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.

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 makes 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 4.6.2.

6 Performance Evaluation

This section has two parts. First, it presents results of experiments to evaluate the benefit of eliminating public-key cryptography from the critical path. Then, it presents an analysis of the cost of proactive recoveries.

6.1 Experimental Setup

All experiments ran with four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. Clients and replicas ran on Dell Precision 410 workstations with Linux 2.2.16-3 (uniprocessor). These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines were connected by a 100 Mb/s switched Ethernet and had 3Com 3C905B interface cards. The switch was an Extreme Networks Summit48 V4.1. The experiments ran on an isolated network.

The interval between checkpoints, K, was 128 requests, which causes garbage collection to occur several times in each experiment. The size of the log, L, was 256. The state partition tree had 4 levels, each internal node had 256 children, and the leaves had 4 KB.

6.2 The cost of Public-Key Cryptography

To evaluate the benefit of using MACs instead of public key signatures, we implemented BFT-PK. Our previous algorithm [6] relies on the extra power of digital signatures to authenticate pre-prepare, prepare, checkpoint, and view-change messages but it can be easily modified to use MACs to authenticate other messages. To provide a fair comparison, BFT-PK is identical to the BFT library but it uses public-key signatures to authenticate these four types of messages. We ran a micro benchmark, and a file system benchmark to compare the performance of services implemented with the two libraries. There were no view changes, recoveries or key changes in these experiments.

6.2.1 Micro-Benchmark

The micro-benchmark compares the performance of two implementations of a simple service: one implementation uses BFT-PK and the other uses BFT. This service has no state and its operations have arguments and results of different sizes but they do nothing. We also evaluated the performance of NO-REP: an implementation of the service using UDP with no replication. We ran experiments to evaluate the latency and throughput of the service. The comparison with NO-REP shows the worst case overhead for our library; in real services, the relative overhead will be lower due to computation or I/O at the clients and servers.

Table 1 reports the latency to invoke an operation when the service is accessed by a single client. The results were obtained by timing a large number of invocations in three separate runs. We report the average of the three runs. The standard deviations were always below 0.5% of the reported value.

system	0/0	0/4	4/0
BFT-PK	59368	59761	59805
BFT	431	999	1046
NO-REP	106	625	630

Table 1: Micro-benchmark: operation latency in microseconds. Each operation type is denoted by *a/b*, where *a* and *b* are the sizes of the argument and result in KB.

BFT-PK has two signatures in the critical path and each of them takes 29.4 ms to compute. The algorithm described in this paper eliminates the need for these signatures. As a result, BFT is between 57 and 138 times faster than BFT-PK. BFT's latency is between 60% and 307% higher than NO-REP because of additional communication and computation overhead. For read-only requests, BFT uses the optimization described in [6] that reduces the slowdown for operations 0/0 and 0/4 to 93% and 25%, respectively.

We also measured the overhead of replication at the client. BFT increases CPU time relative to NO-REP by up to a factor of 5, but the CPU time at the client is only between 66 and 142μs per operation. BFT also increases the number of bytes in Ethernet packets that are sent or

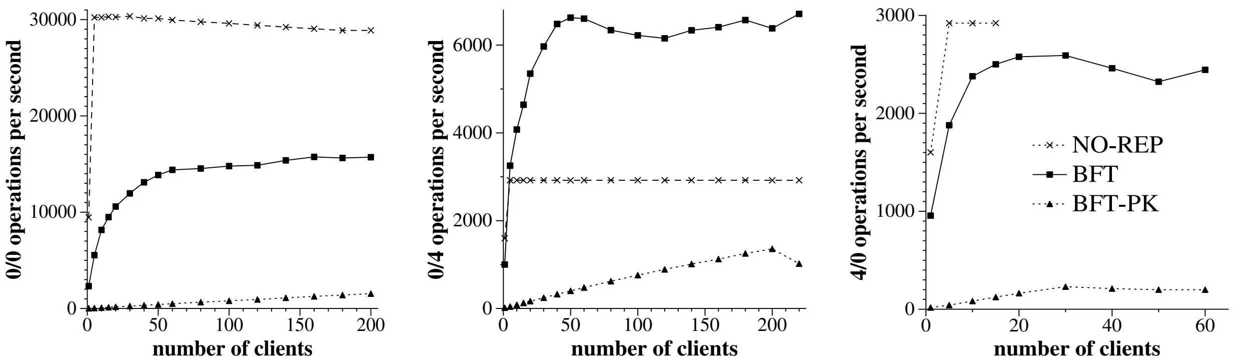

Figure 5: Micro-benchmark: throughput in operations per second.

received by the client: 405% for the 0/0 operation but only 12% for the other operations.

Figure 5 compares the throughput of the different implementations of the service as a function of the number of clients. The client processes were evenly distributed over 5 client machines[2] and each client process invoked operations synchronously, i.e., it waited for a reply before invoking a new operation. Each point in the graph is the average of at least three independent runs and the standard deviation for all points was below 4% of the reported value (except that it was as high as 17% for the last four points in the graph for BFT-PK operation 4/0). There are no points with more than 15 clients for NO-REP operation 4/0 because of lost request messages; NO-REP uses UDP directly and does not retransmit requests.

The throughput of both replicated implementations increases with the number of concurrent clients because the library implements batching [4]. Batching inlines several requests in each pre-prepare message to amortize the protocol overhead. BFT-PK performs 5 to 11 times worse than BFT because signing messages leads to a high protocol overhead and there is a limit on how many requests can be inlined in a pre-prepare message.

The bottleneck in operation 0/0 is the server's CPU; BFT's maximum throughput is 53% lower than NO-REP's due to extra messages and cryptographic operations that increase the CPU load. The bottleneck in operation 4/0 is the network; BFT's throughput is within 11% of NO-REP's because BFT does not consume significantly more network bandwidth in this operation. BFT achieves a maximum aggregate throughput of 26 MB/s in operation 0/4 whereas NO-REP is limited by the link bandwidth (approximately 12 MB/s). The throughput is better in BFT because of an optimization that we described in [6]: each client chooses one replica randomly; this replica's reply includes the 4 KB but the replies of the other replicas only contain small digests. As a result, clients obtain the large replies in parallel from different replicas. We refer the reader to [4] for a detailed analysis of these latency and throughput results.

[2]Two client machines had 700 MHz PIIIs but were otherwise identical to the other machines.

6.2.2 File System Benchmarks

We implemented the Byzantine-fault-tolerant NFS service that was described in [6]. The next set of experiments compares the performance of two implementations of this service: BFS, which uses BFT, and BFS-PK, which uses BFT-PK.

The experiments ran the modified Andrew benchmark [25, 15], which 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. Unfortunately, Andrew is so small for today's systems that it does not exercise the NFS service. So we increased the size of the benchmark by a factor of n as follows: phase 1 and 2 create n copies of the source tree, and the other phases operate in all these copies. We ran a version of Andrew with n equal to 100, Andrew100, and another with n equal to 500, Andrew500. BFS builds a file system inside a memory mapped file [6]. We ran Andrew100 in a file system file with 205 MB and Andrew500 in a file system file with 1 GB; both benchmarks fill 90% of theses files. Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

We also compare BFS and the NFS implementation in Linux, NFS-std. The performance of NFS-std is a good metric of what is acceptable because it is used daily by many users. For all configurations, the actual benchmark code ran at the client workstation using the standard NFS client implementation in the Linux 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 write-back client caching, and allowing attribute caching.

Tables 2 and 3 present the results for these experiments. We report the mean of 3 runs of the benchmark. The standard deviation was always below 1% of the reported averages except for phase 1 where it was as high as 33%. The results show that BFS-PK takes 12 times longer than BFS to run Andrew100 and 15 times longer to run Andrew500. The slowdown is smaller than the one observed with the micro-benchmarks because the

phase	BFS-PK	BFS	NFS-std
1	25.4	0.7	0.6
2	1528.6	39.8	26.9
3	80.1	34.1	30.7
4	87.5	41.3	36.7
5	2935.1	265.4	237.1
total	4656.7	381.3	332.0

Table 2: Andrew100: elapsed time in seconds

client performs a significant amount of computation in this benchmark.

Both BFS and BFS-PK use the read-only optimization described in [6] for reads and lookups, and as a consequence do not set the time-last-accessed attribute when these operations are invoked. This reduces the performance difference between BFS and BFS-PK during phases 3 and 4 where most operations are read-only.

phase	BFS-PK	BFS	NFS-std
1	122.0	4.2	3.5
2	8080.4	204.5	139.6
3	387.5	170.2	157.4
4	496.0	262.8	232.7
5	23201.3	1561.2	1248.4
total	32287.2	2202.9	1781.6

Table 3: Andrew500: elapsed time in seconds

BFS-PK is impractical but BFS's performance is close to NFS-std: it performs only 15% slower in Andrew100 and 24% slower in Andrew500. The performance difference would be lower if Linux implemented NFS correctly. For example, we reported previously [6] that BFS was only 3% slower than NFS in Digital Unix, which implements the correct semantics. The NFS implementation in Linux does not ensure stability of modified data and meta-data as required by the NFS protocol, whereas BFS ensures stability through replication.

6.3 The Cost of Recovery

Frequent proactive recoveries and key changes improve resilience to faults by reducing the window of vulnerability, but they also degrade performance. We ran Andrew to determine the minimum window of vulnerability that can be achieved without overlapping recoveries. Then we configured the replicated file system to achieve this window, and measured the performance degradation relative to a system without recoveries.

The implementation of the proactive recovery mechanism is complete except that we are simulating the secure co-processor, the read-only memory, and the watchdog timer in software. We are also simulating fast reboots. The LinuxBIOS project [22] has been experimenting with replacing the BIOS by Linux. They claim to be able to reboot Linux in 35 s (0.1 s to get the kernel running and 34.9 to execute scripts in `/etc/rc.d`) [22]. This means that in a suitably configured machine we should be able to reboot in less than a second. Replicas simulate a reboot by sleeping either 1 or 30 seconds and calling `msync` to invalidate the service-state pages (this forces reads from disk the next time they are accessed).

6.3.1 Recovery Time

The time to complete recovery determines the minimum window of vulnerability that can be achieved without overlaps. We measured the recovery time for Andrew100 and Andrew500 with 30s reboots and with the period between key changes, T_k, set to 15s.

Table 4 presents a breakdown of the maximum time to recover a replica in both benchmarks. Since the processes of checking the state for correctness and fetching missing updates over the network to bring the recovering replica up to date are executed in parallel, Table 4 presents a single line for both of them. The line labeled *restore state* only accounts for reading the log from disk the service state pages are read from disk on demand when they are checked.

	Andrew100	Andrew500
save state	2.84	6.3
reboot	30.05	30.05
restore state	0.09	0.30
estimation	0.21	0.15
send new-key	0.03	0.04
send request	0.03	0.03
fetch and check	9.34	106.81
total	42.59	143.68

Table 4: Andrew: recovery time in seconds.

The most significant components of the recovery time are the time to save the replica's log and service state to disk, the time to reboot, and the time to check and fetch state. The other components are insignificant. The time to reboot is the dominant component for Andrew100 and checking and fetching state account for most of the recovery time in Andrew500 because the state is bigger.

Given these times, we set the period between watchdog timeouts, T_w, to 3.5 minutes in Andrew100 and to 10 minutes in Andrew500. These settings correspond to a minimum window of vulnerability of 4 and 10.5 minutes, respectively. We also run the experiments for Andrew100 with a 1s reboot and the maximum time to complete recovery in this case was 13.3s. This enables a window of vulnerability of 1.5 minutes with T_w set to 1 minute.

Recovery must be fast to achieve a small window of vulnerability. While the current recovery times are low, it is possible to reduce them further. For example, the time to check the state can be reduced by periodically backing up the state onto a disk that is normally write-protected and by using copy-on-write to create copies of modified pages on a writable disk. This way only the modified pages need to be checked. If the read-only copy of the state is brought up to date frequently (e.g., daily), it will be possible to scale to very large states while achieving even lower recovery times.

6.3.2 Recovery Overhead

We also evaluated the impact of recovery on performance in the experimental setup described in the previous section. Table 5 shows the results. BFS-rec is BFS with proactive recoveries. The results show that adding frequent proactive recoveries to BFS has a low impact on performance: BFS-rec is 16% slower than BFS in Andrew100 and 2% slower in Andrew500. In Andrew100 with 1s reboot and a window of vulnerability of 1.5 minutes, the time to complete the benchmark was 482.4s; this is only 27% slower than the time without recoveries even though every 15s one replica starts a recovery.

The results also show that the period between key changes, T_k, can be small without impacting performance significantly. T_k could be smaller than 15s but it should be substantially larger than 3 message delays under normal load conditions to provide liveness.

system	Andrew100	Andrew500
BFS-rec	443.5	2257.8
BFS	381.3	2202.9
NFS-std	332.0	1781.6

Table 5: Andrew: recovery overhead in seconds.

There are several reasons why recoveries have a low impact on performance. The most obvious is that recoveries are staggered such that there is never more than one replica recovering; this allows the remaining replicas to continue processing client requests. But it is necessary to perform a view change whenever recovery is applied to the current primary and the clients cannot obtain further service until the view change completes. These view changes are inexpensive because a primary multicasts a view-change message just before its recovery starts and this causes the other replicas to move to the next view immediately.

7 Related Work

Most previous work on replication techniques assumed benign faults, e.g., [17, 23, 18, 19] or a synchronous system model, e.g., [28]. Earlier Byzantine-fault-tolerant systems [26, 16, 20], including the algorithm we described in [6], could guarantee safety only if fewer 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 and frequently; it can tolerate any number of faults if fewer than $1/3$ of the replicas become faulty within a window of vulnerability, which can be made small under normal load conditions with low impact on performance.

In a previous paper [6], we described a system that tolerated Byzantine faults in asynchronous systems and performed well. This paper extends that work by providing recovery, a state transfer mechanism, and a new view change mechanism that enables both recovery and an important optimization — the use of MACs instead of public-key cryptography.

Rampart [26] and SecureRing [16] provide group membership protocols that can be used to implement recovery, but only in the presence of benign faults. These approaches cannot be guaranteed to 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, the algorithms rely on messages signed by replicas even after they are removed from the group and there is no way to prevent attackers from impersonating removed replicas that they controlled.

The problem of efficient state transfer has not been addressed by previous work on Byzantine-fault-tolerant replication. We present an efficient state transfer mechanism that enables frequent proactive recoveries with low performance degradation.

Public-key cryptography was the major performance bottleneck in previous systems [26, 16] despite the fact that these systems include sophisticated techniques to reduce the cost of public-key cryptography at the expense of security or latency. They cannot use MACs instead of signatures because they rely on the extra power of digital signatures to work correctly: signatures allow the receiver of a message to prove to others that the message is authentic, whereas this may be impossible with MACs. The view change mechanism described in this paper does not require signatures. It allows public-key cryptography to be eliminated, except for obtaining new secret keys. This approach improves performance by up to two orders of magnitude without loosing security.

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 [24]. This concept has previously been applied in synchronous systems to secret-sharing schemes [13], threshold cryptography [14], and more recently secure information storage and retrieval [10] (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.

8 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 described here improves the security and robustness against software errors of previous systems by recovering replicas proactively and frequently. It can tolerate any number of faults provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability. This window can be 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 detects the corruption of the state of a recovering replica.

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 performance close to that of an unreplicated service. Furthermore, they show that the window of vulnerability can be made very small: 1.5 to 10 minutes with only 2% to 27% degradation in performance.

Acknowledgments

We would like to thank Kyle Jamieson, Rodrigo Rodrigues, Bill Weihl, and the anonymous referees for their helpful comments on drafts of this paper. We also thank the Computer Resource Services staff in our laboratory for lending us a switch to run the experiments and Ted Krovetz for the UMAC code.

References

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

[2] J. Black et al. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO*, 1999.

[3] R. Canetti, S. Halevi, and A. Herzberg. Maintaining Authenticated Communication in the Presence of Break-ins. In *ACM Conference on Computers and Communication Security*, 1997.

[4] M. Castro. *Practical Byzantine Faul Tolerance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000. In preparation.

[5] 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.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[7] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. Technical Report 2000-03, University of Arizona, 2000.

[8] S. Floyd et al. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6), 1995.

[9] S. Forrest et al. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 1997.

[10] J. Garay et al. Secure Distributed Storage and Retrieval. *Theoretical Computer Science*, to appear.

[11] L. Gong. A Security Risk of Depending on Synchronized Clocks. *Operating Systems Review*, 26(1):49–53, 1992.

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

[13] A. Herzberg et al. Proactive Secret Sharing, Or: How To Cope With Perpetual Leakage. In *Advances in Cryptology - CRYPTO*, 1995.

[14] A. Herzberg et al. Proactive Public Key and Signature Systems. In *ACM Conference on Computers and Communication Security*, 1997.

[15] J. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988.

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

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

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

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

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

[21] D. Maziéres et al. Separating Key Management from File System Security. In *ACM Symposium on Operating System Principles*, 1999.

[22] Ron Minnich. The Linux BIOS Home Page. http://www.acl.lanl.gov/linuxbios, 2000.

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

[24] R. Ostrovsky and M. Yung. How to Withstand Mobile Virus Attacks. In *ACM Symposium on Principles of Distributed Computing*, 1991.

[25] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX Summer*, 1990.

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

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

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