Appears as Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, June 1999

Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography

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

We have developed a *practical* state-machine replication algorithm that tolerates Byzantine faults: it works correctly in asynchronous systems like the Internet and it incorporates several optimizations that improve the response time of previous algorithms by more than an order of magnitude. This paper describes the most important of these optimizations. It explains how to modify the base algorithm to eliminate the major performance bottleneck in previous systems — public-key cryptography. The optimization replaces public-key signatures by vectors of message authentication codes during normal operation, and it overcomes a fundamental limitation on the power of message authentication codes relative to digital signatures — the inability to prove that a message is authentic to a third party. As a result, authentication is more than two orders of magnitude faster while providing the same level of security.

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.

We have developed a *practical* algorithm for state-machine replication [14, 29] that tolerates Byzantine faults. The algorithm is described in [5]. It offers both liveness and safety provided at most $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas are faulty. This means that clients eventually receive replies to their requests and those replies are correct according to linearizability [12, 4].

Unlike previous algorithms [29, 25, 13], ours works correctly in asynchronous systems like the Internet, and it incorporates important optimizations that enable it to outperform previous systems by more than an order of magnitude [5]. This paper describes the most important of these optimizations in detail. It explains how to modify the base algorithm to eliminate the major performance bottleneck in previous systems — the cost of using public-key cryptography to produce digital signatures.

The time to perform public-key cryptography operations to generate and verify signatures is cited as the major bottleneck [24, 17, 13] in state-machine replication algorithms designed for practical application. The optimization described in this paper replaces digital signatures by vectors of message authentication codes during normal operation. It uses digital signatures only for view changes that occur when a replica fails and are likely to be infrequent. For the same level of security and typical service configurations, our authentication scheme is more than two orders of magnitude faster than one using public-key signatures.

Message authentication codes are widely used. What is interesting in the work described here is that it overcomes a fundamental limitation of message authentication codes relative to digital signatures — the inability to prove that a message is authentic to a third party. Previous state-machine replication algorithms [24, 26, 13] (as well as the base version of our algorithm described in [5]) rely on this property of digital signatures for correctness. We explain how to modify our algorithm to overcome this problem while retaining the same communication performance during normal case operation and the same resiliency. Our solution to the problem takes advantage of the bound of $\lfloor \frac{n-1}{3} \rfloor$ on the number of faulty replicas (which is also required by the algorithms that use digital signatures) and the fact that correct replicas agree on an order for requests.

This work is part of our research to produce practical Byzantine fault tolerance algorithms and to demonstrate their practicality by implementing real systems. Our initial results are very promising. We have implemented a Byzantine-fault-tolerant NFS file system and it performs less than 3% slower than a standard, unreplicated implementation of NFS [5].

The rest of the paper is organized as follows. Section 2 presents an overview of our system model and lists our assumptions. Section 3 describes the problem

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. Miguel Castro was partially supported by a PRAXIS XXI fellowship.

solved by the algorithm and states correctness conditions. Section 4 describes the algorithm using public-key signatures. These sections also appeared in [5]. They are repeated here for completeness and because the version of the algorithm with public-key signatures is easier to understand, but can be skipped by a reader that is familiar with the algorithm in [5]. Section 5 discusses the changes to the algorithm that allow us to avoid public-key cryptography during normal operation. Section 6 discusses related work. Our conclusions are presented in Section 7.

2 System Model

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

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the restriction mentioned below. We assume independent node failures. For this assumption to be true in the presence of malicious attacks, some steps need to be taken, e.g., each node should run different implementations of the service code and operating system and should have a different root password and a different administrator. It is possible to obtain different implementations from the same code base [23] and for low degrees of replication one can buy operating systems from different vendors. N-version programming, where different teams of programmers produce different implementations, is another option for some services.

We use cryptographic techniques to prevent spoofing and replays and to detect corrupted messages. Our messages contain public-key signatures [28], message authentication codes [30], and message digests produced by collision-resistant hash functions [27]. We denote a message m signed by node i as $\langle m \rangle_{\sigma_i}$ and the digest of message m by $D(m)$. We follow the common practice of signing a digest of a message and appending it to the plaintext of the message rather than signing the full message ($\langle m \rangle_{\sigma_i}$ should be interpreted in this way.) All replicas know the public keys of other replicas and clients to verify signatures. Clients also know the public keys of replicas.

We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely. We also assume that the adversary (and the faulty nodes it controls) are computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques mentioned above. For example, the adversary cannot produce a valid signature of a non-faulty node, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties [28, 30, 27].

3 Service Properties

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

The algorithm provides both *safety* and *liveness* assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. Safety means that the replicated service satisfies linearizability [12] (modified to account for Byzantine-faulty clients [4]): it behaves like a centralized implementation that executes operations atomically one at a time.

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

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

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

The resiliency of our algorithm is optimal: $3f+1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty (see [2] for a proof.)

4 The Algorithm

Our algorithm is a form of *state machine* replication [14, 29]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. We denote the set of replicas by $\mathcal{R}$ and identify each replica using an integer in $\{0, ..., |\mathcal{R}|-1\}$. For simplicity, we assume $|\mathcal{R}| = 3f+1$ where f is the maximum number of replicas that may be faulty.

The replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. Views are numbered consecutively. The primary of a view is replica p such that $p = v \; mod \; |\mathcal{R}|$, where v is the view number. View changes are carried out when it appears that the primary has failed. Viewstamped Replication [21] and Paxos [15] used a similar approach to tolerate benign faults.

The algorithm works roughly as follows:

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

Like all state machine replication techniques [29], we impose two requirements on replicas: they must be *deterministic* (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result) and they must start in the same state. Given these two requirements, the algorithm ensures the safety property by guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures*.

Section 5 explains how to avoid using public-key signatures in the common case. The remainder of this section describes a simplified version of the algorithm that uses digital signatures for message authentication and omits a description of message retransmissions. A detailed formalization of the algorithm using the I/O automaton model [16] is presented in [4].

4.1 The Client

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

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

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

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

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

4.2 Normal-Case Operation

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

When the primary, p, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas.

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

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

A backup accepts a pre-prepare message provided:

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

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

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

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

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

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

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

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

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

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

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

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

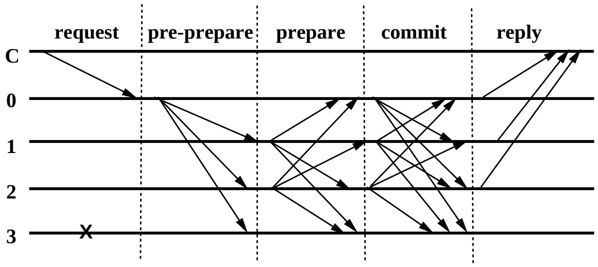

Figure 1: Normal Case Operation

4.3 Garbage Collection

This section discusses the mechanism used to discard messages from the log. For the safety condition to hold, messages must be kept in a replica's log until it knows that the requests they concern have been executed by at least $f+1$ replicas and it can prove this to others in view changes.

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

A replica maintains several logical copies of the service state: the last stable checkpoint, zero or more checkpoints that are not stable, and a current state. Copy-on-write techniques can be used to reduce the space overhead to store the extra copies of the state, as was done in [5].

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

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

Computing the proofs is efficient because the digest can be computed using incremental cryptography [1] (as was done in [5]), and proofs are generated rarely.

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

4.4 View Changes

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

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

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

1. The primary determines the sequence number *min-s* of the latest stable checkpoint in $\mathcal{V}$ and the highest sequence number *max-s* in a prepare message in $\mathcal{V}$.
2. The primary creates a new pre-prepare message for view $v+1$ for each sequence number n between *min-s* and *max-s*. There are two cases: (1) there is at least one set in the $\mathcal{P}$ component of some view-change message in $\mathcal{V}$ with sequence number n, or (2) there is no such set. In the first case, the primary creates a new message $\langle \text{PRE-PREPARE}, v+1, n, d \rangle_{\sigma_p}$, where d is the request digest in the pre-prepare message for sequence number n with the highest view number in $\mathcal{V}$. In the second case, it creates a new pre-prepare message $\langle \text{PRE-PREPARE}, v+1, n, d^{null} \rangle_{\sigma_p}$, where d^{null} is the digest of a special *null* request; a null request goes through the protocol like other requests, but its execution is a no-op. (Paxos [15] used a similar technique to fill in gaps.)

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

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

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

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

4.5 Safety

This section sketches the proof that the algorithm provides safety; details can be found in [4]. As discussed earlier, the algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally.

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

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

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

5 Avoiding Public-Key Cryptography

The algorithm in the previous section performs poorly because it requires a large number of digital signature operations for each client request. This section describes a modified version of the algorithm that eliminates this performance bottleneck by replacing public-key signatures by vectors of message authentication codes during normal operation.

The section begins with a description of the new authentication mechanism and a discussion of its benefits. Then, it explains how to modify the algorithm to overcome the inability of message authentication codes to prove authenticity of a message to a third party.

5.1 Authenticators

The optimized algorithm uses vectors of message authentication codes to authenticate messages. We call these vectors *authenticators*.

Message authentication codes use symmetric cryptography to authenticate the communication between two parties that share a secret session key. They work as follows: the sender of a message m computes a small bit string, which is a function of m and the key it shares with the receiver, and appends this string (the message authentication code or MAC) to the message; the receiver can check the authenticity of m by computing the MAC in the same way and comparing it to the one appended to the message.

To compute message authentication codes, each replica and each (active) client shares a secret session key with each replica. There is actually a pair of session keys for each pair of replicas i and j: $K_{i,j}$ is used for messages sent from i to j, and $K_{j,i}$ is used for messages sent from j to i. Each replica has in addition a secret session key for each client; this key is used for communication in both directions.

We use a pair of keys for communication between replicas, rather than a single session key, to allow replicas to change independently the keys that they use to verify incoming messages. The use of a different session key for each replica also prevents the failure of a node from affecting the authentication of messages sent between pairs that do not include the faulty node.

The session keys are established (and refreshed) dynamically using a corrected version of the Denning-Sacco protocol [7]. For example, replica i chooses a new (random) key $K_{j,i}$ to be used by replica j to compute MACs for messages sent by j to i. Then, it sends $\langle \text{NEW-KEY}, i, j, \langle K_{j,i} \rangle_{\epsilon_j}, t \rangle_{\sigma_i}$ to j and stops accepting messages from j that are not authenticated using $K_{j,i}$. Here, $\langle K_{j,i} \rangle_{\epsilon_j}$ denotes encryption with j's public-key, and t is a timestamp. The timestamps are monotonically increasing and j rejects any new-key message with a timestamp lower than the timestamp of the last such message it accepted from i; this prevents replay attacks. If replica j accepts the message, it caches $K_{j,i}$ and uses it to authenticate messages it sends to i.

The session key establishment between clients and replicas is similar. A client, c, establishes a new session key, $K_{c,i}$ with a replica, i, by sending the message $\langle \text{NEW-KEY}, c, i, \langle K_{c,i} \rangle_{\epsilon_i}, t \rangle_{\sigma_c}$ to i. Replica i rejects any new-key message with a timestamp lower than the timestamp of the last such message it accepted from c. If replica i accepts the message, it caches $K_{c,i}$ and uses it to check whether requests sent by c are authentic and to authenticate the replies to requests by c. Replica i will accept messages authenticated with $K_{c,i}$ until c establishes a new session key, explicitly closes the session, or a predefined key lifetime expires.

These session keys are used to authenticate the messages in the algorithm. The digital signature in a reply message is replaced by a single MAC, which is sufficient because these messages have a single intended recipient. We denote the MAC produced using a session key $K_{i,j}$ for some message m as $\mu_{i,j}(m)$. The signatures in all other messages (including client requests but excluding view changes) are replaced by authenticators. An authenticator generated by i for a message m has an entry for every replica j other than i; each entry is the MAC, $\mu_{i,j}(m)$. We use the notation $\langle m \rangle_{\alpha_i}$ to represent a message m with an authenticator generated by i appended.

In our current implementation [5], session keys are 16 bytes. We compute message authentication codes by applying MD5 to the concatenation of the message with the secret key. Rather than using the 16 bytes of the final MD5 digest, we use only the 10 least significant bytes. This truncation has the obvious advantage of reducing the size of MACs and it also improves their resilience to certain attacks [22]. This is a variant of the secret suffix method [30], which is secure as long as MD5 is collision resistant [22, 8].

Discussion

The advantage of MACs over digital signatures is that they can be computed three orders of magnitude faster. For example, a 200MHz Pentium Pro takes 43ms to generate a 1024-bit modulus RSA signature of an MD5 digest and 0.6ms to verify the signature [31], whereas it takes only 10.3μs to compute the MAC of a 64-byte message on the same hardware in our implementation. There are other public-key cryptosystems that generate signatures faster, e.g., elliptic curve public-key cryptosystems, but signature verification is slower [31] and in our algorithm each signature is verified many times.

There may be doubts that authenticators benefit from a similar performance advantage over digital signatures because some of the costs of managing authenticators grow linearly with the number of replicas. The next paragraphs argue that for typical configurations authenticators are more than two orders of magnitude faster than digital signatures.

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

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

Another potential problem is session key management. Since session keys can be established easily as we explained, the only possible problem is the cost of storing session keys. Clients only store n keys; this account for less than 208 bytes for $n \leq 13$. Replicas store $2n + c$ session keys where c is the number of active clients. This is also reasonable because replicas can limit the number of active clients, e.g., it takes approximately 1 MB to store the keys for 50000 active clients (assuming 8 byte client identifiers and in spite of our large session keys.)

5.2 Modified Algorithm

The real problem with authenticators is that they lack the following important property of digital signatures: if some replica accepts a signed message as authentic, all other replicas will also accept the message as authentic.

With authenticators, the receiver of a message may be unable to prove its authenticity to a third party. The version of our algorithm described in Section 4 relies on the additional power of signatures for correctness; replicas use digital signatures to prove to other replicas that requests prepared and that checkpoints are correct. Rampart [25] and SecureRing [13] also relied on signatures for correctness.

This section explains how to modify the algorithm to overcome this fundamental limitation of message authentication codes. The modified algorithm is based on three key observations. First, authenticators are sufficient to guarantee safety in the normal case. A replica only needs to prove to others that messages it accepted are authentic during view changes, or when an out-of-date replica is brought up to date by transferring a checkpoint.

Second, replicas can use authenticators in the normal case and obtain digital signatures lazily only when they need to send a proof to another replica. If a replica accepted l messages with authenticators from different replicas, it can later ask those replicas to send signatures for their messages; it is guaranteed to obtain $l - f$ replies because of the bound on the number of faults. Unfortunately, l is only guaranteed to be at least $2f + 1$ in a system with a total of $3f + 1$ replicas. Therefore, a replica may obtain only $f+1$ signatures. This is sufficient to prove the correctness of a checkpoint but not that a request prepared.

The third observation is that it is not necessary to prove that a request prepared in view changes, i.e., it is not necessary to send $2f + 1$ signed pre-prepare/prepare messages within a view-change message. Instead, $f + 1$ signed messages are sufficient if the view change protocol is modified to take advantage of the invariant that non-faulty replicas agree on the sequence numbers assigned to requests that prepare in the same view.

The rest of this section explain how these observations enable a correct algorithm that uses signatures only for view changes.

5.2.1 Normal Case Operation

We will now describe the modifications to the protocol during normal case operation. These modifications are simple except that the protocol needs to defend against faulty clients that create partially incorrect authenticators (as discussed at the end of this section.)

Clients interact with the service as described before by sending requests and receiving replies. Requests have the same format except that the client's signature is replaced by an authenticator and there is an additional MAC appended to the message. This MAC authenticates the authenticator to allow the primary, p, to detect tampering. The new format is: $\langle r = \langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}, \mu_{c,p}(r) \rangle$. The only difference in reply messages is that the replica's signature is replaced by a MAC computed with the key shared by the replica and the client.

When the primary p receives a client request m, it checks whether $\mu_{c,p}$ and its MAC in m's authenticator are both correct. If they are not, it ignores the request. Otherwise, it assigns a sequence number n to m and multicasts a pre-prepare message with m piggybacked to all the backups. The only difference in the pre-prepare message format is that the primary's signature is replaced by an authenticator. Since $\mu_{c,p}$ guarantees the authenticity of m's authenticator, all backups will be able to authenticate m provided the client is non-faulty (and there is no tampering while the pre-prepare message is in transit.)

Backups accept the pre-prepare message under the conditions that were described in Section 4 except that they only accept the pre-prepare if the corresponding MACs in the authenticators of the pre-prepare and request messages are correct. The rest of the protocol remains unchanged. The only difference is that replicas' signatures in prepare and commit messages are replaced by authenticators.

Faulty clients can attack this protocol by sending requests with partially incorrect authenticators, i.e., authenticators with correct MACs for some replicas but incorrect ones for others. There are three types of attacks.

First, a faulty client could potentially leave non-faulty replicas in an inconsistent state by sending partially incorrect authenticators. This is prevented by having replicas check the authenticator in the client request when they receive a pre-prepare. If replica i receives a pre-prepare for request m with a valid authenticator from the primary, the pre-prepare is entered in the log and i will not send a prepare for a different request with the same view and sequence number. But i only sends a prepare message if it can authenticate m. If later m commits at i (i.e., *committed-local* becomes true,) i can execute m even if it is unable to authenticate it because at least one correct replica was able to authenticate m.

Second, a faulty client can send a request with an authenticator that has a correct MAC for the primary but incorrect MACs for sufficiently many replicas that it cannot prepare at $2f + 1$ replicas. This attack does not compromise the safety of the algorithm but it can affect the performance because it allows faulty clients to force view changes. To avoid this problem, we extended the protocol to allow the primary to abort these requests. The mechanism used is similar to a view change and therefore we omit a description here. The main differences are that it does not change views (i.e., it does not change who the primary is) and, since it concerns a single request, it is significantly less expensive than a view change. After a primary aborts a client request, it will only accept requests from that client that are signed with the client's private key. This prevents faulty clients from causing the abort protocol to run frequently.

Third, a client can force a view change by sending a retransmitted request to at least $f + 1$ backups with an authenticator that contains a correct MAC for each of those backups but an incorrect MAC for the primary. To prevent this problem, backups only accept retransmitted requests with digital signatures.

5.2.2 Garbage Collection

There are only two changes to the garbage collection mechanism: the replica's signature in checkpoint messages is replaced by an authenticator; and replicas collect $2f + 1$ matching checkpoint messages before they can discard information from the log (whereas only $f + 1$ were necessary in the algorithm in Section 4.) The $2f + 1$ messages are necessary to ensure that the replica can obtain a proof of correctness for its stable checkpoint when required. During normal case operation, replicas never obtain proofs of correctness for their stable checkpoints but view change messages sent by the replica need to include these proofs. The proofs may also be needed to bring another replica up-to-date by transferring a checkpoint.

The following mechanism allows a replica i to obtain a proof for its stable checkpoint or for a later checkpoint: i signs its own checkpoint message and asks the other replicas for signed checkpoint messages. This is done by multicasting a *check-sign* request to these replicas with the form $\langle \text{CHECK-SIGN}, n, i \rangle_{\alpha_i}$ (where n is the sequence number of the checkpoint for which the signature is requested.) When a non-faulty replica j receives a check-sign message, it replies with a set of checkpoint messages with its signature. This set includes all the checkpoint messages j sent that are still in its log and have sequence number greater than or equal to n.

Replica i waits until it has $f + 1$ matching signed checkpoint messages from different replicas for its stable checkpoint or a checkpoint with a higher sequence number. If it does not receive enough messages, it keeps retransmitting the check-sign request. Replica i is guaranteed to eventually have the $f + 1$ signed messages for the following reasons.

1. Replica i starts with $2f + 1$ matching checkpoint messages from different replicas.
2. There is a bound f on the number of faults.
3. Non-faulty replicas retain information about checkpoint messages they sent in their log until they have $2f + 1$ matching checkpoint messages from different replicas with a higher sequence number.

5.2.3 View Changes

The algorithm in Section 4 requires replicas to include proofs for checkpoints and prepared requests in view change messages. A proof for a checkpoint consists of $f + 1$ signed checkpoint messages and a proof for a prepared request consists of a signed pre-prepare and $2f$ signed prepare messages. The previous section explains how to obtain the proof for a checkpoint lazily in the optimized algorithm. Unfortunately, it is not possible to obtain lazily a proof that a request prepared; a replica may be unable to obtain more than $f + 1$ signed pre-prepare/prepare messages. The optimized algorithm overcomes this problem by modifying the view change protocol such that it is sufficient to include $f + 1$ signed pre-prepare/prepare messages in view-change messages.

View-Change Messages

View-change messages are sent as before and they retain the same format: $\langle \text{VIEW-CHANGE}, v + 1, n, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$. The only difference is that each set $\mathcal{P}_m$ in $\mathcal{P}$ contains only $f + 1$ signed pre-prepare/prepare messages rather than $2f + 1$.

Replicas obtain the $f + 1$ signed checkpoint messages in $\mathcal{C}$ as described in the previous section and use a similar mechanism to obtain the signed messages for each $\mathcal{P}_m$. For each request m such that *prepared*(m, v, n', i) is *true* (for some v and $n' > n$), i obtains $\mathcal{P}_m$ by signing any matching pre-prepare/prepare message it sent and by multicasting a message with the form $\langle \text{PREPARE-SIGN}, v, n', D(m), i \rangle_{\alpha_i}$ to the other replicas. When a non-faulty replica j receives a prepare-sign message, it checks whether its stable checkpoint has a sequence number greater than or equal to n'; if it has, j replies as if the prepare-sign message was a check-sign message with sequence number n'. Else, if j sent a matching pre-prepare or prepare message that is still in its log, it sends back a signed copy of that message. Otherwise, the prepare-sign message is ignored.

Replica i retransmits the prepare-sign until it has $f + 1$ signed pre-prepare/prepare messages matching the prepare-sign or $f + 1$ matching signed checkpoint messages with a sequence number greater than or equal to n'. This will eventually happen because:

1. *prepared*$(m, v, n', i) =$ *true*;
2. there is a bound f on the number of faults; and
3. Non-faulty replicas retain information about checkpoint and pre-prepare/prepare messages they sent in their log until they have $2f + 1$ matching checkpoint messages from different replicas with a higher sequence number.

If i obtains a proof of correctness for a checkpoint with sequence number greater than or equal to n', it includes the proof in $\mathcal{C}$ and prunes $\mathcal{P}$ accordingly.

New-View Messages

The problem with this protocol is that it is possible for replicas to produce correct view-change messages that *conflict* whereas this was impossible in the algorithm described in Section 4. Two view-change messages *conflict* if they both contain a set of $f + 1$ signed pre-prepare/prepare messages for the same view and sequence number and each set corresponds to a different request. Since it is possible for one of the requests to have committed, the primary must be able to choose the correct view-change message from a set of conflicting ones.

The optimized protocol overcomes this problem in a subtle but surprisingly simple way. It takes advantage of the following invariant: non-faulty replicas agree on the sequence numbers assigned to requests that prepare in the same view. The invariant implies that view-changes sent by non-faulty replicas never conflict and the bound f on the number of faulty replicas implies that the primary will eventually have $2f + 1$ view-change messages from non-faulty replicas. Therefore, the primary p of view $v + 1$ waits until it has $2f + 1$ correct, non-conflicting view-change messages for view $v + 1$ signed by different replicas (including its own.) Then, it builds a $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_p}$ message (as before) and multicasts this message to all other replicas. (It is actually not necessary to include $\mathcal{O}$ in the message because it can be computed from $\mathcal{V}$ by the backups.)

The rest of the protocol proceeds as before. Note that a replica can still obtain missing requests and checkpoints from the replicas whose $f + 1$ signatures appear in the corresponding set in some view-change message; this is guaranteed to work because at least one of those replicas is non-faulty.

The modified view change protocol verifies the key correctness conditions listed in Section 4.5. In particular, any pre-prepare message in $\mathcal{O}$ is guaranteed to be consistent with what happened in previous views. This is true because $f + 1$ signed pre-prepare/prepare messages for a request m in a view v with sequence number n ensure that at least one non-faulty replica accepted a pre-prepare for m with sequence number n in view v. And, if a request committed in a previous view, it will be proposed by at least $f + 1$ non-faulty replicas in their view-change messages and at least one of these will be part of the new-view message.

Optimization

This protocol can be optimized by not sending signed pre-prepare/prepare and checkpoint messages in view changes. Instead, the $\mathcal{C}$ component of a view-change message sent by a replica i can be replaced by a digest of the checkpoint and the $\mathcal{P}$ component can be a set of tuples of the form $\langle n, v, d\rangle$ (where n is the sequence number of the request, v is the latest view in which the request prepared at i, and d is a digest of the request.) If the set of view-change messages in the new-view message contains $f + 1$ elements with the same checkpoint digest for the same sequence number, there is no need to obtain $f + 1$ signed checkpoint messages to certify the checkpoint value. Otherwise, the primary will ask one of the proponents of the checkpoint to obtain such checkpoint messages. Signatures of pre-prepare and prepare messages are avoided in a similar way. We expect this optimization to avoid most signature generation operations.

6 Related Work

Most previous work on replication techniques ignored Byzantine faults or assumed a synchronous system model (e.g., [14, 21, 15, 29, 6, 10].) But some agreement and consensus algorithms tolerate Byzantine faults in asynchronous systems (e.g,[2, 3, 19].) However, they do not provide a complete solution for state machine replication, and furthermore, most of them were designed to demonstrate theoretical feasibility and are too slow to be used in practice. Our algorithm during normal-case operation is similar to the Byzantine agreement algorithm in [2], which also does not use public-key cryptography. But their algorithm is insufficient to implement state-machine replication; it only guarantees that non-faulty processes agree on the messages sent by a primary. It does not handle view changes on primary failures (i.e., does not provide liveness if the primary fails), garbage collection, or client authentication. Handling these issues is the most interesting and subtle contribution of the optimization presented in this paper.

The two systems that are most closely related to our work are Rampart [24, 25, 26, 17] and SecureRing [13]. They implement state machine replication but are more than an order of magnitude slower than our system and, most importantly, they rely on synchrony assumptions for safety. The time to perform public-key cryptography operations to generate and verify signatures is cited as the major latency [24] and throughput bottleneck [17] in Rampart and SecureRing. It is interesting to note that this is true despite the fact that these systems include sophisticated techniques to reduce the cost of public-key cryptography. Rampart [24] uses short-term public/private key pairs in a 300-bit modulus RSA cryptosystem and refreshes these keys with periodic view changes. The modification to Rampart described in [17] and SecureRing [13] improve throughput at the expense of latency by chaining signatures to reduce the number of signatures per request.

This paper describes a technique to eliminate the public-key cryptography bottleneck completely without increasing the number of message delays to execute a request. The number of message delays introduced by our algorithm between the moment the client sends a request and receives a reply is only 4 for read-write requests and 2 for read-only requests [5]. Rampart requires 5 message delays to execute client requests and SecureRing requires significantly more. Furthermore, our efficient authentication scheme allows us to use hardware multicast to transmit protocol messages rather than relying on point-to-point authenticated channels as in Rampart or penalizing latency in favor of throughput as in SecureRing.

Phalanx [18, 20] applies quorum replication techniques [11] to achieve Byzantine fault-tolerance in asynchronous systems. This work does not provide generic state machine replication; instead, it offers a data repository with operations to read and write individual variables and to acquire locks. There are no published performance numbers for Phalanx but we believe our algorithm is faster because it has fewer message delays in the critical path and because Phalanx relies on public-key cryptography during normal case operation.

7 Conclusion

This paper described an algorithm for Byzantine-fault-tolerant state machine replication that eliminates the major performance bottleneck in previous systems — public-key cryptography. The algorithm replaces public-key signatures by vectors of message authentication codes during normal operation. It uses public-key signatures only for view changes that occur when the primary fails and are likely to be rare. This new technique decreases the cost of authentication by more than two orders of magnitude relative to previous systems while preserving the same level of security.

Our algorithm is interesting because it overcomes a fundamental limitation of message authentication codes relative to digital signatures — the inability to prove that a message is authentic to a third party. We explain how to overcome this limitation by taking advantage of the assumed bound of $\lfloor \frac{n-1}{3} \rfloor$ on the number of faulty replicas (which is also required by the algorithms that use digital signatures) and the fact that correct replicas agree on the sequence numbers of requests that prepare in the same view. Previous systems for Byzantine-fault-tolerant state machine replication relied on the extra power of public-key digital signatures for correctness.

Byzantine-fault-tolerant replication is a promising solution to increase the availability and integrity of current systems. However, previous algorithms were not practical because they were too slow or relied on synchrony assumptions for correctness. This paper is an important step towards the development of practical algorithms; a Byzantine-fault-tolerant NFS file system implemented using the optimized algorithm described in this paper performs less than 3% slower than a standard, unreplicated implementation of NFS [5].

References

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

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

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

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

[5] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

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

[7] D. E. Denning and G. M. Sacco. Timestamps in Key Distribution Protocols. *Comm. of the ACM*, 24(8):533–536, August 1981.

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

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

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

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

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

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

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

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

[16] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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

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

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

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

[21] 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.

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

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

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

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

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

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

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

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

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

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