

# Practical Byzantine Fault Tolerance and Proactive Recovery

MIGUEL CASTRO

Microsoft Research

and

BARBARA LISKOV

MIT Laboratory for Computer Science

---

Our growing reliance on online services accessible on the Internet demands highly available systems that provide correct service without interruptions. Software bugs, operator mistakes, and malicious attacks are a major cause of service interruptions and they can cause arbitrary behavior, that is, Byzantine faults. This article describes a new replication algorithm, BFT, that can be used to build highly available systems that tolerate Byzantine faults. BFT can be used in practice to implement real services: it performs well, it is safe in asynchronous environments such as the Internet, it incorporates mechanisms to defend against Byzantine-faulty clients, and it recovers replicas proactively. The recovery mechanism allows the algorithm to tolerate any number of faults over the lifetime of the system provided fewer than 1/3 of the replicas become faulty within a small window of vulnerability. BFT has been implemented as a generic program library with a simple interface. We used the library to implement the first Byzantine-fault-tolerant NFS file system, BFS. The BFT library and BFS perform well because the library incorporates several important optimizations, the most important of which is the use of symmetric cryptography to authenticate messages. The performance results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated. This supports our claim that the BFT library can be used to build practical systems that tolerate Byzantine faults.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*; D.4.3 [**Operating Systems**]: File Systems Management; D.4.5 [**Operating Systems**]: Reliability—*Fault tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls; authentication; cryptographic controls*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Security, Reliability, Algorithms, Performance, Measurement

Additional Key Words and Phrases: Byzantine fault tolerance, state machine replication, proactive recovery, asynchronous systems, state transfer

---

This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. Part of this work was done while M. Castro was with the MIT Laboratory for Computer Science and during this time he was partially supported by Praxis XXI and Gulbenkian fellowships.

Authors' addresses: M. Castro, Microsoft Research, 7 J. J. Thomson Avenue, Cambridge CB3 0FB, UK; email: mcastro@microsoft.com; B. Liskov, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2002 ACM 0734-2071/02/1100-0398 \$5.00

## 1. INTRODUCTION

We are increasingly dependent on services provided by computer systems and our vulnerability to computer failures is growing as a result. We would like these systems to be highly available: they should work correctly and they should provide service without interruptions.

There is a large body of research on replication techniques to implement highly available systems. The problem is that most research on replication has focused on techniques that tolerate benign faults (e.g., Alsberg and Day [1976], Gifford [1979], Oki and Liskov [1988], Lamport [1989], and Liskov et al. [1991]): these techniques assume components fail by stopping or by omitting some steps. They may not provide correct service if a single faulty component violates this assumption. Unfortunately, this assumption is not valid because malicious attacks, operator mistakes, and software errors are common causes of failure and they can cause faulty nodes to exhibit arbitrary behavior, that is, Byzantine faults. The growing reliance of industry and government on computer systems provides the motif for malicious attacks and the increased connectivity to the Internet exposes these systems to more attacks. Operator mistakes are also cited as one of the main causes of failure [Murphy and Levidow 2000]. In addition, the number of software errors is increasing due to the growth in size and complexity of software.

Techniques that tolerate Byzantine faults [Pease et al. 1980; Lamport et al. 1982] provide a potential solution to this problem because they make no assumptions about the behavior of faulty processes. There is a significant body of work on agreement and replication techniques that tolerate Byzantine faults. However, most earlier work (e.g., Canetti and Rabin [1992], Reiter [1996], Malkhi and Reiter [1996b], Garay and Moses [1998], and Khilstrom et al. [1998]) either concerns techniques that are too inefficient to be used in practice, or relies on assumptions that can be invalidated easily by an attacker. For example, it is dangerous to rely on *synchrony* [Lamport 1984] for safety in the Internet, that is, to rely on bounds on message delays and process speeds. An attacker may compromise the correctness of a service by delaying nonfaulty nodes or the communication between them until the bounds are exceeded. Such a denial-of-service attack is generally easier than gaining control over a non-faulty node.

This article describes BFT, a new algorithm for state machine replication [Lamport 1978; Schneider 1990] that offers both liveness and safety provided at most  $\lfloor (n - 1)/3 \rfloor$  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 [Herlihy and Wing 1987; Castro and Liskov 1999a].

BFT is the first Byzantine-fault-tolerant, state machine replication algorithm that is safe in asynchronous systems such as the Internet: it does not rely on any synchrony assumption to provide safety. In particular, it never returns bad replies even in the presence of denial-of-service attacks. Additionally, it guarantees liveness provided message delays are bounded eventually. The service may be unable to return replies when a denial-of-service attack is active but clients are guaranteed to receive replies when the attack ends.

Since BFT is a state machine replication algorithm, it has the ability to replicate services with complex operations. This is an important defense against Byzantine-faulty clients: operations can be designed to preserve invariants on the service state, to offer narrow interfaces, and to perform access control. BFT provides safety regardless of the number of faulty clients and the safety property ensures that faulty clients are unable to break these invariants or bypass access controls. Algorithms that only offer reads, writes, and synchronization primitives (e.g., Malkhi and Reiter [1998b]) are more vulnerable to Byzantine-faulty clients; they rely on clients to order and synchronize reads and writes correctly in order to enforce invariants.

We also describe a proactive recovery mechanism for BFT that recovers replicas periodically even if there is no reason to suspect that they are faulty. This allows the replicated system 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. 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. The window of vulnerability can be made very small (e.g., a few minutes) under normal conditions with a low impact on performance. Our mechanism provides detection of denial-of-service attacks aimed at increasing the window and it also detects when the state of a replica is corrupted by an attacker.

BFT incorporates a number of important optimizations that allow the algorithm to perform well so that it can be used in practice. The most important optimization is the use of symmetric cryptography to authenticate messages. Public key cryptography, which was cited as the major latency [Reiter 1994] and throughput [Malkhi and Reiter 1996a] bottleneck in previous systems, is used only to exchange the symmetric keys. Other optimizations reduce the communication overhead: the algorithm uses only one message round trip to execute read-only operations and two to execute read-write operations, and it uses batching under load to amortize the protocol overhead for read-write operations over many requests. The algorithm also uses optimizations to reduce protocol overhead as the operation argument and result sizes increase. Additionally, the article describes efficient techniques to garbage collect protocol information, and to transfer state to bring replicas up to date; these are necessary to build practical services that tolerate Byzantine faults.

BFT has been implemented as a generic program library with a simple interface. The BFT library can be used to provide Byzantine-fault-tolerant versions of different services. The article describes the BFT library and explains how it was used to implement a real service: the first Byzantine-fault-tolerant distributed file system, BFS, which supports the NFS protocol.

The article presents a performance analysis of the BFT library and BFS. The experimental results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated. These results were obtained in configurations with four and seven replicas that can tolerate one and two Byzantine faults, respectively. They support our claim that the BFT library can be used to implement practical Byzantine-fault-tolerant systems.

The rest of the article is organized as follows. Section 2 presents our system model and assumptions, and Section 3 describes the problem solved by the algorithm and states correctness conditions. The algorithm without recovery is described informally in Section 4 and formally in the Appendix. The proactive recovery mechanism is presented in Section 5. Section 6 describes optimizations and implementation techniques that are important for implementing a practical solution for replication in the presence of Byzantine faults. The implementation of the BFT library and BFS is presented in Section 7. Section 8 presents a detailed performance analysis for the BFT library and BFS. Section 9 discusses related work. Finally, our conclusions and some directions for future work appear in Section 10.

## 2. SYSTEM MODEL

A replicated service is implemented by  $n$  replicas that execute operations requested by clients. Replicas and clients run in different nodes in a distributed system and are connected by a network.

BFT implements a form of state machine replication [Lamport 1978; Schneider 1990] that allows replication of services that perform arbitrary computations provided they are deterministic, that is, replicas must produce the same sequence of results when they process the same sequence of operations.

Replicas use a cryptographic hash function  $D$  to compute message digests, and they use message authentication codes (MACs) to authenticate all messages including client requests [Schneier 1996]. 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  $i$ . Each replica also shares a single secret key with each client; this key is used to authenticate communication in both directions. These session keys can be established and refreshed dynamically using the mechanism described in Section 5.2.2 or any other key exchange protocol.

Messages that are sent point-to-point to a single recipient contain a single MAC; 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  $k_{i,j}$ . Messages that are multicast to all the replicas 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.

BFT assumes very little from the nodes and the network. We use a Byzantine failure model; that is, faulty nodes may behave arbitrarily. (Replicas and clients are correct if they follow the algorithm in Section 4.) The network that connects nodes may fail to deliver messages, delay them, duplicate them, or deliver them out of order. Therefore, we allow for a very strong adversary that can control faulty nodes and the network in order to cause the most damage to the replicated service. For example, it can coordinate faulty nodes, delay messages, or inject new messages.

We rely only on the following assumptions: the first two are assumptions on the behavior of nodes, required both for safety and for liveness, and the last one

is an assumption on the behavior of the network, required only for liveness. The **proactive recovery mechanism** relies on additional (realistic) assumptions that are described in Section 5.1.

### Bound on Faults

We assume a bound  $f = \lfloor (n-1)/3 \rfloor$  on the number of faulty replicas. In Section 5, we describe a proactive recovery mechanism that enables the algorithm to tolerate any number of faults over the lifetime of the system provided at most  $f$  replicas fail in any small window of vulnerability. But the proactive recovery mechanism requires additional assumptions.

There is little benefit in using the BFT library or any other replication technique when there is a strong positive correlation between the failure probabilities of the replicas; the probability of violating the bound on the number of faults is not significantly larger than the probability of a single fault in this case. For example, our approach cannot mask a software error that occurs at all replicas at the same time. But the BFT library can mask nondeterministic software errors, which seem to be the most persistent [Gray 2000] since they are the hardest to detect.

One can increase the benefit of replication further by taking 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, administrator attacks or mistakes, attacks involving physical access to the replicas, 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 service implementations to improve resilience to software bugs and attacks that exploit software bugs. The version of the BFT library described in this article does not allow software diversity but we have recently developed an extension to the library that does [Rodrigues et al. 2001].

### Strong Cryptography

We also assume that the adversary is computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques mentioned above. We assume the attacker cannot forge MACs: if  $i$  and  $j$  are non-faulty nodes and they never generated  $\langle m \rangle_{\mu_{ij}}$ , the adversary is unable to generate  $\langle m \rangle_{\mu_{ij}}$  for any  $m$ . We also assume that the cryptographic hash function is collision resistant: the adversary is unable to find two distinct messages  $m$  and  $m'$  such that  $D(m) = D(m')$ . These assumptions are probabilistic but they are believed to hold with high probability for the cryptographic primitives we use [Black et al. 1999; Rivest 1992]. Therefore, we assume that they hold with probability one in the rest of the text.

The algorithm does not rely on any form of cryptographic signature attached to messages to prove that they are authentic to a third party. Therefore, it can be modified easily to rely only on point-to-point authenticated channels. This can be done simply by sending copies of a message (without MACs) over multiple

channels instead of multicasting the message (with MACs). It is also possible to modify the algorithm not to use a cryptographic hash function by replacing the hash of a message by the value of the message. The resulting algorithm is secure against adversaries that are not computationally bound provided the authenticated channels can be made secure against such adversaries (which may be possible using, for example, quantum cryptography [Bennett et al. 1992]). But since most authenticated channel implementations rely on computational bounds on the adversary, we present an efficient version of the algorithm that relies on this assumption.

In addition, if we were only concerned with nonmalicious faults (e.g., software errors), it would be possible to relax the assumptions about the cryptographic primitives and use weaker, more efficient constructions.

#### Weak Synchrony (Only for Liveness)

Let  $\text{delay}(t)$  be 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 (where the sender keeps retransmitting the message until it is received, and both sender and destination are correct). We assume that  $\text{delay}(t)$  has an asymptotic upper bound. Currently, we assume that  $\text{delay}(t) = o(t)$  but the bounding function can be changed easily.

### 3. SERVICE PROPERTIES

BFT provides both safety and liveness properties [Lamport 1977] assuming no more than  $\lfloor (n-1)/3 \rfloor$  replicas are faulty over the lifetime of the system.

The safety property is a form of linearizability [Herlihy and Wing 1987]: the replicated service behaves as a centralized implementation that executes operations atomically one at a time. The original definition of linearizability does not work with Byzantine-faulty clients. We describe our modified definition of linearizability in Appendix B.

The resilience of BFT is optimal: at least  $3f + 1$  replicas are necessary to provide the safety and liveness properties under our assumptions when up to  $f$  replicas are faulty. To understand the bound on the number of faulty replicas, consider a replicated service that implements a mutable variable with read and write operations. To provide liveness, the service may have to return a reply before the request is received by more than  $n - f$  replicas, since  $f$  replicas might be faulty and not responding. Therefore, the service may reply to a write request after the new value is written only to a set  $W$  with  $n - f$  replicas. If later a client issues a read request, it may receive a reply based on the state of a set  $R$  with  $n - f$  replicas.  $R$  and  $W$  may have only  $n - 2f$  replicas in common. Additionally, it is possible that the  $f$  replicas that did not respond are not faulty and, therefore,  $f$  of those that responded might be faulty. As a result, the intersection between  $R$  and  $W$  may contain only  $n - 3f$  nonfaulty replicas. It is impossible to ensure that the read returns the correct value unless  $R$  and  $W$  have at least one nonfaulty replica in common; therefore  $n > 3f$ .

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 nonfaulty clients. In particular, if the service operations are designed to preserve some invariants on the service state, faulty clients cannot break those invariants. This is an important defense against Byzantine-faulty clients that is enabled by BFT's ability to implement an arbitrary abstract data type [Liskov and Zilles 1975].

Some algorithms only provide primitives to read a single variable or to write a single variable; they are more vulnerable to Byzantine-faulty clients because they rely on clients to implement complex service operations using these primitives. Even when systems provide mutual exclusion operations to group reads and writes (e.g., Malkhi and Reiter [1998b, 2000]), they rely on clients to order and group these primitive operations correctly to enforce the invariants required by the service operations. For example, creating a file requires updates to metadata information. In BFT, this operation can be implemented to enforce metadata invariants such as ensuring the file is assigned a new inode. In algorithms that rely on clients to implement complex operations, a faulty client will be able to write metadata information and violate important invariants; for example, it could assign the inode of another file to the newly created file.

The invariants enforced by service operations may be insufficient to guard against faulty clients; for example, in a file system a faulty client can write garbage data to some shared file. Therefore, we further 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. Since operations can be arbitrarily complex, the access control policy can be specified at an abstract level (e.g., the ability to create files in a directory). This contrasts with systems where access control policy can only specify the ability to read or write each object (e.g., Malkhi and Reiter [1998b, 2000]). Additionally, the algorithm allows services to change access permissions dynamically while still ensuring linearizability. This provides a mechanism to recover from attacks by faulty clients.

BFT 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 [Fischer et al. 1985]. We guarantee liveness (i.e., clients eventually receive replies to their requests), provided at most  $\lfloor (n-1)/3 \rfloor$  replicas are faulty and  $\text{delay}(t)$  does not grow faster than  $t$  indefinitely. This is a rather weak synchrony assumption that is likely to be true in any real system provided network faults are eventually repaired and denial-of-service attacks eventually stop, yet it enables us to circumvent the impossibility result.

Our algorithm does not address the problem of fault-tolerant privacy: a faulty replica may leak information to an attacker. It is not yet practical to offer fault-tolerant privacy in the general case because service operations may perform arbitrary computations using their arguments and the service state; replicas need this information in the clear to execute such operations efficiently. But it is easy to ensure privacy by having clients encrypt arguments that are opaque to service operations.

Algorithms that tolerate Byzantine faults are subtle. Therefore, it is important to specify them formally and to prove their correctness. We wrote a

formal specification for a simplified version of the algorithm and proved its safety [Castro 2001]. The simplified version is identical to the one described in this article except that messages are authenticated using public key cryptography. Recently, Lamport [2001] formalized a simplified version of the algorithm described in this article (without public key cryptography) and argued its correctness.

#### 4. THE BFT ALGORITHM

This section describes the algorithm without proactive recovery. We omit some important optimizations and details related to message retransmissions. The optimizations are explained in Section 6 and message retransmissions are explained in Castro [2001]. We present a formalization of the algorithm in the Appendix.

##### 4.1 Overview

Our algorithm builds on previous work on state machine replication [Lamport 1978; Schneider 1990]. The service is modeled as a state machine that is replicated across different nodes in a distributed system. Each replica maintains the service state and implements the service operations. Clients send requests to execute operations to the replicas and BFT ensures that all nonfaulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all nonfaulty 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 in state machine replication is ensuring nonfaulty replicas execute the same requests in the same order. Like Viewstamped Replication [Oki and Liskov 1988] and Paxos [Lamport 1989], our algorithm uses a combination of primary-backup [Alsberg and Day 1976] and quorum replication [Gifford 1979] techniques to order requests. But it tolerates Byzantine faults whereas Paxos and Viewstamped Replication only tolerate benign faults.

In a primary-backup mechanism, replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. The primary picks the ordering for execution of operations requested by clients. It does this by assigning the next available sequence number to a request and sending this assignment to the backups. But the primary may be faulty: it may assign the same sequence number to different requests, stop assigning sequence numbers, or leave gaps between sequence numbers. Therefore the backups check the sequence numbers assigned by the primary and use timeouts to detect when it stops. They trigger view changes to select a new primary when it appears that the current one has failed.

The algorithm ensures that request sequence numbers are *dense*, that is, no sequence numbers are skipped but when there are view changes some sequence numbers may be assigned to null requests whose execution is a no-op.



To order requests correctly despite failures, we rely on quorums [Gifford 1979]. We can use any Byzantine dissemination quorum system construction [Malkhi and Reiter 1998a]. These quorums have two important properties.

- **Intersection:** any two quorums have at least one correct replica in common.
- **Availability:** there is always a quorum available with no faulty replicas.

These properties enable the use of quorums as a reliable memory for protocol information. Replicas write information to a quorum and they collect *quorum certificates*, which are sets with one message from each element in a quorum saying that it stored the information. These certificates are proof that the information has been reliably stored and will be reflected in later reads. Reads from the reliable memory obtain the information stored by all the elements in a quorum and pick the latest piece of information.

We also use *weak certificates*, which are sets with at least  $f + 1$  messages from different replicas. Weak certificates prove that at least one correct replica stored the information. Every step in the protocol is justified by a certificate.

We denote the set of replicas by  $\mathcal{R}$  and identify each replica using an integer in  $\{0, \dots, |\mathcal{R}| - 1\}$ . For simplicity, we assume  $|\mathcal{R}| = 3f + 1$  where  $f$  is the maximum number of replicas that may be faulty. We choose the primary of a view to be replica  $p$  such that  $p = v \bmod |\mathcal{R}|$ , where  $v$  is the view number and views are numbered consecutively. Currently, our quorums are just sets with at least  $2f + 1$  replicas.

## 4.2 The Client

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

Replicas accept the request and add it to their log provided they can authenticate it. Request execution is ordered 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  $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$  where  $v$  is the current view number,  $t$  is the timestamp of the corresponding request,  $i$  is the replica number, and  $r$  is the result of executing the requested operation.

The client waits for a **weak certificate** with  $f + 1$  replies with valid MACs from different replicas, and with the same  $t$  and  $r$ , before accepting the result  $r$ . Since at most  $f$  replicas can be faulty, this ensures that the result is valid. We call this certificate the *reply certificate*.

If the client does not receive a reply certificate soon enough, it retransmits the request. If the request has already been processed, the replicas simply retransmit the reply; replicas remember the last reply message they sent to each client to enable this retransmission. If the primary does not assign a valid sequence number to the request, it will eventually be suspected to be faulty by enough replicas to cause a view change.

We assume that the client waits for one request to complete before sending the next one but it is not hard to change the protocol to allow a client to make asynchronous requests, yet preserve ordering constraints on them.

The next paragraphs discuss scalability with the number of clients. First, replicas share a secret key with each client. This could create a scalability problem with a large number of clients. We avoid this problem as follows. Replicas only share secret keys with active clients and they limit the number of active clients. New session keys can be established as described in Section 5.2.2 when the set of active clients changes. Key information does not take a large amount of space even with a large bound on the number of active clients. For example, with 50,000 active clients this information uses less than 1 MB of space assuming 16-byte keys and 8-byte client identifiers.

Additionally, replicas need to remember the 8-byte timestamp of the last request executed by each client to ensure exactly once semantics. But since timestamps are small and timestamps of inactive clients can be stored on disk, this should not cause a significant scalability problem. However, replicas also store the last reply message sent to each client to enable retransmissions. This is impractical if replies are large and there are a large number of clients. The implementation can trade off the ability to retransmit lost reply messages for scalability. Replicas can bound the amount of space used to store this information by discarding the oldest replies. If a replica receives a request whose reply has been discarded, it informs the client that the request has been executed but the reply is no longer available. We believe that the bound and the frequency of request retransmissions can be made sufficiently large that this is unlikely to happen. Furthermore, the client may be able to query the service and obtain a reply after this happens.

#### 4.3 Normal Case Operation

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 provides an overview of the algorithm in the normal case of no faults.

The state of each replica includes the state of the service, a *message log* containing messages the replica has accepted or sent, and an integer denoting the replica's current view. We describe how to truncate the log in Section 4.4. The state can be kept in volatile memory; it does not need to be stable.

When the primary  $p$  receives a request  $m_{\alpha_c} = \langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}$  from a client, it assigns a sequence number  $n$  to  $m$  provided it can authenticate the request. Then it multicasts a PRE-PREPARE message with the assignment to the backups and inserts this message in its log. The message has the form  $\langle \text{PRE-PREPARE}, v, n, D(m) \rangle_{\alpha_p}$ , where  $v$  indicates the view in which the message is being sent and  $D(m)$  is  $m$ 's digest.

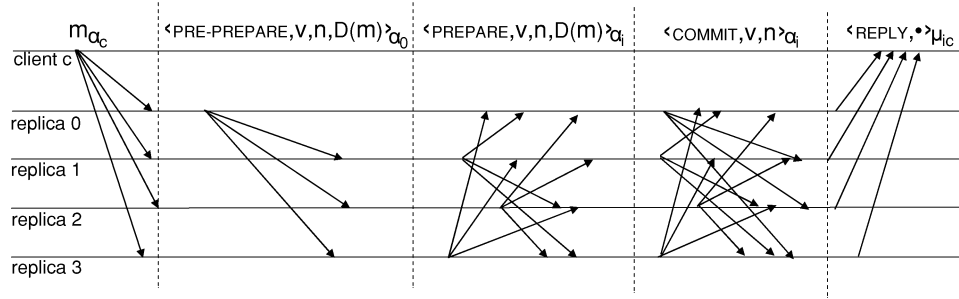


Fig. 1. Normal case operation: the primary (replica 0) assigns sequence number  $n$  to request  $m$  in its current view  $v$  and multicasts a PRE-PREPARE message with the assignment. If a backup agrees with the assignment, it multicasts a matching PREPARE message. When a replica receives messages that agree with the assignment from a quorum, it sends a COMMIT message. Replicas execute  $m$  after receiving COMMIT messages from a quorum.

Like PRE-PREPARES, the PREPARE and COMMIT messages sent in the other phases also contain  $n$  and  $v$ . A replica only accepts one of these messages provided that it is in view  $v$ ; that it can verify the authenticity of the message; and that  $n$  is between a low water mark  $h$  and a high water mark  $H$ . The last condition is necessary to enable garbage collection and to 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 4.4.

A backup  $i$  accepts the PRE-PREPARE message provided (in addition to the conditions above) it has not accepted a PRE-PREPARE for view  $v$  and sequence number  $n$  containing a different digest. If a backup  $i$  accepts the PRE-PREPARE and it has request  $m$  in its log, it enters the *prepare* phase by multicasting a  $\langle \text{PREPARE}, v, n, D(m), i \rangle_{\alpha_i}$  message with  $m$ 's digest to all other replicas; in addition, it adds both the PRE-PREPARE and PREPARE messages to its log. Otherwise, it does nothing. The PREPARE message signals that the backup agreed to assign sequence number  $n$  to  $m$  in view  $v$ . We say that a request is *pre-prepared* at a particular replica if the replica sent a PRE-PREPARE or PREPARE message for the request.

Then each replica collects messages until it has a quorum certificate with the PRE-PREPARE and **2f** matching PREPARE messages for sequence number  $n$ , view  $v$ , and request  $m$ . We call this certificate the *prepared certificate* and we say that the replica *prepared* the request. This certificate proves that a quorum has agreed to assign number  $n$  to  $m$  in  $v$ . **The protocol guarantees that it is not possible to obtain prepared certificates for the same view and sequence number and different requests.**

It is interesting to reason why this is true because it illustrates one use of quorum certificates. Assume that it were false and there existed two distinct requests  $m$  and  $m'$  with prepared certificates for the same view  $v$  and sequence number  $n$ . Then the quorums for these certificates would have at least one non-faulty replica in common. This replica would have sent PRE-PREPARE or PREPARE messages agreeing to assign the same sequence number to both  $m$  and  $m'$  in the same view. Therefore,  $m$  and  $m'$  would not be distinct, which contradicts our assumption.

This ensures that replicas agree on a total order for requests in the same view but it is not sufficient to ensure a total order for requests across view changes. Replicas may collect prepared certificates in different views with the same sequence number and different requests. The commit phase solves this problem as follows. Each replica  $i$  multicasts  $\langle \text{COMMIT}, v, n, i \rangle_{\alpha_i}$  saying it has the prepared certificate and adds this message to its log. Then each replica collects messages until it has a quorum certificate with  $2f + 1$  COMMIT messages for the same sequence number  $n$  and view  $v$  from different replicas (including itself). We call this certificate the *committed certificate* and say that the request is *committed* by the replica when it has both the prepared and committed certificates.

After the request is committed, the protocol guarantees that the request has been prepared by a quorum; that is, there is a quorum which knows that a quorum has accepted to assign number  $n$  to a request in view  $v$ . New primaries ensure information about committed requests is propagated to new views by reading prepared certificates from a quorum and selecting the sequence number assignments in the certificates for the latest views. The view change protocol is described in detail in Section 4.5.

Each replica  $i$  executes the operation requested by the client when  $m$  is committed and the replica has executed all requests with lower sequence numbers. This ensures that all nonfaulty replicas execute requests in the same order as is required to provide safety. After executing the requested operation, replicas send a reply to the client. To guarantee exactly once semantics, replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client.

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.

It is possible for a request's authenticator to have both correct and incorrect MACs if the client is faulty, or the request was corrupted in the network. Therefore it is necessary to design the protocol to ensure that replicas agree on whether a request is authentic. Otherwise, this problem could lead to safety and liveness violations. BFT solves this problem by generalizing the mechanism used to verify the authenticity of requests; a replica  $i$  can authenticate a request if the MAC for  $i$  in the request's authenticator is correct, or  $i$  has  $f + 1$  PRE-PREPARE or PREPARE messages with the request's digest in its log. The first condition is usually sufficient but the second condition prevents the system from deadlocking if a request with a partially correct authenticator commits at some correct replica.

#### 4.4 Garbage Collection

This section discusses the garbage collection mechanism that prevents message logs from growing without bound. Replicas must discard information about requests that have already been executed from their logs. But a replica cannot simply discard messages when it executes the corresponding requests because it could discard a prepared certificate that would later be necessary to ensure

safety. Instead, the replica must first obtain a proof that its state is correct. Then, it can discard messages corresponding to requests whose execution is reflected in the state.

Generating these proofs after executing every operation would be expensive. Instead, they are generated periodically, when a request with a sequence number divisible by the *checkpoint period*  $K$  is executed (e.g.,  $K = 128$ ). We refer to the states produced by the execution of these requests as *checkpoints* and we say that a checkpoint with a proof is a *stable checkpoint*.

When replica  $i$  produces or fetches 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 last stable checkpoint, zero or more checkpoints that are not stable, and the current state. This is necessary to ensure that the replica has both the state and the matching proof for its stable checkpoint. Section 6.2 describes how we manage checkpoints and transfer state between replicas efficiently.

Each replica collects messages until it has a quorum certificate with  $2f + 1$  CHECKPOINT messages (including its own) authenticated by different replicas with the same sequence number  $n$  and digest  $d$ . We call this certificate the *stable certificate*; it ensures other replicas will be able to obtain a *weak certificate* proving that the stable checkpoint is correct if they need to fetch it. At this point, the checkpoint with sequence number  $n$  is 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 the maximum number of consecutive sequence numbers for which the replica will log information. It is obtained by multiplying  $K$  by a small constant factor (e.g., 2) that is big enough so that it is unlikely for replicas to stall waiting for a checkpoint to become stable.

#### 4.5 View Changes

The view change protocol provides liveness by allowing the system to make progress when the primary fails. The protocol must also preserve safety: it must ensure that nonfaulty replicas agree on the sequence numbers of committed requests across views.

The basic idea behind the protocol is for the new primary to read information about stable and prepared certificates from a quorum and to propagate this information to the new view. Since any two quorums intersect, the primary is guaranteed to obtain information that accounts for all requests that committed in previous views and all stable checkpoints. The rest of this section describes a simplified view change protocol that may require *unbounded space*. We present a modification to the protocol in Castro [2001] that eliminates the problem.

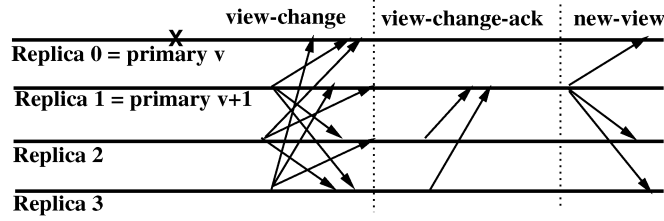


Fig. 2. View-change protocol: the primary for view  $v$  (replica 0) fails causing a view change to view  $v + 1$ .

### Data Structures

Replicas record information about what happened in earlier views. This information is maintained in two sets,  $\mathcal{P}$  and  $\mathcal{Q}$ . These sets only contain information for **sequence numbers between the current low and high water marks in the log**. 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 empty while the system is running normally. Replicas also store the requests corresponding to entries in these sets.

$\mathcal{P}$  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  $i$  collected a prepared certificate for a request with digest  $d$  with number  $n$  in view  $v$  and no request **prepared** at  $i$  in a later view with the same number.

$\mathcal{Q}$  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  $\langle n, d, v \rangle$** , meaning that  $i$  pre-prepared a request with digest  $d$  with number  $n$  in view  $v$  and that request did not **pre-prepare** at  $i$  in a later view with the same number.

### View-Change Messages

Figure 2 illustrates the view-change protocol from view  $v$  to view  $v + 1$ . When a backup  $i$  suspects the primary for view  $v$  is faulty, it enters view  $v + 1$  and multicasts a  **$\langle \text{VIEW-CHANGE}, v + 1, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, i \rangle_{a_i}$**  message to all replicas. Here  $h$  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 the sets described above. **These sets are updated before sending the VIEW-CHANGE message** using the information in the log, as explained in Figure 3. Once the VIEW-CHANGE message has been sent,  $i$  removes PRE-PREPARE, PREPARE, and COMMIT messages from its log. The number of tuples in  $\mathcal{Q}$  may grow without bound if the algorithm changes views repeatedly without making progress. In Castro [2001], we describe a modification to the algorithm that bounds the size of the  $\mathcal{Q}$  by a constant. **It is interesting to note that VIEW-CHANGE messages do not include PRE-PREPARE, PREPARE, or CHECKPOINT messages.**

### View-Change-Ack Messages

Replicas collect VIEW-CHANGE messages for  $v + 1$  and send acknowledgments for them to  $v + 1$ 's primary,  $p$ . Replicas only accept these VIEW-CHANGE

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 in view  $v$  then
    if  $\exists \langle n, d', v' \rangle \in \mathcal{P}$ 
      remove  $\langle n, d', v' \rangle$  from  $\mathcal{P}$ 
    add  $\langle n, d, v \rangle$  to  $\mathcal{P}$ 
  if request number  $n$  with digest  $d$  is pre-prepared in view  $v$  then
    if  $\exists \langle n, d, v' \rangle \in \mathcal{Q}$ 
      remove  $\langle n, d, v' \rangle$  from  $\mathcal{Q}$ 
    add  $\langle n, d, v \rangle$  to  $\mathcal{Q}$ 

```

Fig. 3. Computing  $\mathcal{P}$  and  $\mathcal{Q}$ .

messages if all the information in their  $\mathcal{P}$  and  $\mathcal{Q}$  components is for view numbers less than or equal to  $v$ . 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.

#### 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. These VIEW-CHANGE-ACK messages together with the VIEW-CHANGE message it received and the VIEW-CHANGE-ACK it could have sent form a quorum certificate. We call it the *view-change certificate*. Each entry in  $\mathcal{S}$  is for a different replica.

The new primary uses the information in  $\mathcal{S}$  and the decision procedure sketched in Figure 4 to choose a checkpoint and a set of requests. This procedure runs each time the primary receives new information, for example, when it adds a new message to  $\mathcal{S}$ . We use the notation  $m.x$  to indicate component  $x$  of message  $m$  where  $x$  is the name we used for the component when defining the format for  $m$ 's message type.

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 (because they have a weak certificate) and that have numbers higher than the low water mark in the log of at least  $f + 1$  nonfaulty replicas. The last condition is necessary for liveness; 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  $n$  between  $h$  and  $h + L$  (where  $L$  is the size of the log). If a request  $m$  committed in a previous view, the primary must select  $m$ . If such a request exists, it is guaranteed to be the only one that satisfies conditions A1 and A2. Condition A1 ensures that the primary selects the request that some replica in a quorum claims to have prepared in the latest view  $v$ , and A2 ensures



```

let  $\mathcal{C} = \{ \langle n, d \rangle \mid \exists S, S' \subseteq \mathcal{S} : |S| > 2f \wedge |S'| > f \\
\wedge \forall m \in S : m.h \leq n \wedge \forall m \in S' : \langle n, d \rangle \in m.\mathcal{C} \}$ 

if  $\exists \langle h, d \rangle \in \mathcal{C} : \forall \langle n', d' \rangle \in \mathcal{C} : n' \leq h$  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  $m' \in \mathcal{S} :$ 

 $m'.h < n \wedge \forall \langle n', d', v' \rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$ 
        A2.  $\exists f + 1$  messages  $m' \in \mathcal{S} : \exists \langle n', d', v' \rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$ 
        then select the request with digest  $d$  for number  $n$ 

    B. else if  $\exists 2f + 1$  messages  $m \in \mathcal{S}$  such that  $m.h < n \wedge m.\mathcal{P}$  has no entry for  $n$ 
        then select the null request for number  $n$ 
    
```

Fig. 4. Decision procedure at the primary.

that the request could prepare in view  $v$  because it was pre-prepared by at least one correct replica in  $v$  or a later view.

If there is a quorum of replicas that did not prepare any request with sequence number  $n$  (condition B), no request committed with number  $n$ . Therefore, the primary selects a special null request that goes through the protocol as a regular request but whose execution is a no-op. (Paxos [Lamport 1989] used a similar technique to fill in gaps.)

The decision procedure ends when the primary has selected a request for each number. This may require waiting for more than  $n - f$  messages but a primary is always able to complete the decision procedure once it receives all VIEW-CHANGE messages sent by nonfaulty replicas for its view. After deciding, the primary multicasts a NEW-VIEW message to the other replicas with its decision:  $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{X} \rangle_{\alpha_p}$ . 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. The VIEW-CHANGES in  $\mathcal{V}$  are the new-view certificate.

### New-View Message Processing

The primary updates its state to reflect the information in the NEW-VIEW message. It obtains any requests in  $\mathcal{X}$  that it is missing and if it does not have the checkpoint with sequence number  $h$ , it also initiates the protocol to fetch the missing state (see Section 6.2.2). When it has all requests in  $\mathcal{X}$  and the checkpoint with sequence number  $h$  is stable, it records in its log that the requests are pre-prepared in view  $v + 1$ .

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 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. Since the primary only includes a VIEW-CHANGE message in



$S$  after obtaining a matching view-change certificate, at least  $f + 1$  nonfaulty 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 nonfaulty 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 if these messages support the decisions reported by the primary by carrying out the decision procedure in Figure 4. 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, normal case operation resumes.

**4.5.1 Correctness.** We now argue informally that the view-change protocol preserves safety and that it is live.

*Safety.* We start by sketching a proof of the following claim.

If a request  $m$  commits with sequence number  $n$  at some correct replica in view  $v$  then no other request commits with  $v$  and  $n$  at another correct replica, and the decision procedure in Figure 4 will not choose a distinct request for sequence number  $n$  in any view  $v' > v$ .

This claim implies that after a request commits in view  $v$  with sequence number  $n$  no distinct request can pre-prepare at any correct replica with the same sequence number for views later than  $v$ . Therefore, correct replicas agree on a total order for requests because they never commit distinct requests with the same sequence number.

The proof is by induction on the number of views between  $v$  and  $v'$ . If  $m$  committed at some correct replica  $i$ ,  $i$  received COMMIT messages from a quorum of replicas  $Q$ , saying that they prepared the request with sequence number  $n$  and view  $v$ . By the quorum intersection property, distinct requests cannot prepare at a correct replica with the same view and sequence number. Therefore the claim is true in the base case  $v' = v$ .

For the inductive step ( $v' > v$ ), assume by contradiction that the decision procedure chooses a request  $m' \neq m$  for sequence number  $n$  in  $v'$ . This implies that either condition A1 or condition B must be true. By the quorum intersection property, there must be at least one VIEW-CHANGE message from a correct replica  $j \in Q$  with  $h < n$  in any quorum certificate used to satisfy conditions A1 or B.

From the inductive hypothesis and the procedure to compute  $\mathcal{P}$  described in Figure 3,  $j$ 's VIEW-CHANGE message for  $v'$  must include  $\langle n, D(m), v_c \rangle$  in its  $\mathcal{P}$  component with  $v_c \geq v$  (because  $j$  did not garbage collect information for sequence number  $n$ ). Therefore condition B cannot be true. But condition A1 can be true if a VIEW-CHANGE message from a faulty replica includes  $\langle n, D(m'), v_f \rangle$  in its  $\mathcal{P}$  component with  $v_f > v_c$ ; condition A2 prevents this problem. Condition A2 is true only if there is a VIEW-CHANGE message from a correct replica with  $\langle n, D(m'), v'_c \rangle$  in its  $\mathcal{Q}$  component such that  $v'_c \geq v_f$ . Since  $D(m') \neq D(m)$  (with

high probability), the inductive hypothesis implies that  $v'_c \leq v$ . Therefore,  $v_f \leq v$  and conditions A1 and A2 cannot both be true, which finishes the proof.

*Liveness.* To provide liveness, replicas must move to a new view if they are unable to execute a request. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute or when backups detect that the primary is faulty. 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.

We now argue informally that the algorithm is live. We start by arguing that a correct primary will be able to send a NEW-VIEW message provided it has enough time before correct replicas change to the next view. Then we explain how the algorithm maximizes the amount of time available to complete view changes and process some new request.

Assume by contradiction that a correct primary with unbounded time is unable to reach a decision using the procedure in Figure 4. We start by showing that there is at least one checkpoint that satisfies the conditions in the decision procedure. The primary will be able to make progress by choosing this checkpoint or any other checkpoint that satisfies these conditions. Let  $h_c$  be the sequence number of the latest checkpoint that is stable at some correct replica. Since there are at least  $2f + 1$  correct replicas and at least  $f + 1$  correct replicas have the checkpoint with number  $h_c$ , the primary will be able to choose the value  $h_c$  for  $h$ . If necessary to make progress, replicas will be able to fetch any checkpoint chosen by the primary because at least one correct replica has the checkpoint.

For each sequence number  $n$  between  $h$  and  $h + L$ , we argue that the primary can choose a request that satisfies conditions A or B. The cases are: (1) some correct replica prepared a request with sequence number  $n$ ; or (2) there is no such replica. In Case (1), condition A1 will be verified because there are  $2f + 1$  nonfaulty replicas and nonfaulty replicas never prepare different requests for the same view and sequence number; A2 will also be satisfied since a request that prepares at a nonfaulty replica pre-prepares at at least  $f + 1$  nonfaulty replicas. Furthermore, condition A2 implies that there is at least one correct replica with the request that vouches for its authenticity. Therefore any replica that is missing the chosen request can fetch it and can believe that it is authentic. In Case (2), condition B will eventually be satisfied because there are  $2f + 1$  correct replicas that by assumption did not prepare any request with sequence number  $n$ .

It is important to maximize the period of time when at least  $2f + 1$  nonfaulty replicas are in the same view and one of them is the primary. In addition, we can adjust timeouts to ensure that this period of time increases exponentially until some operation executes. We achieve these goals by several means.

First, to avoid starting a view change too soon, a replica that multicasts a VIEW-CHANGE message for view  $v + 1$  waits for  $2f + 1$  VIEW-CHANGE messages for view  $v + 1$  before starting its timer. Then, it starts its timer to expire after some

time  $T$ . If the timer expires before it receives a valid `NEW-VIEW` message for  $v + 1$  or before it executes a request in the new view that it had not executed previously, it starts the view change for view  $v + 2$  but this time it will wait  $2T$  before starting a view change for view  $v + 3$ .

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

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

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

Our implementation guarantees *fairness*: it ensures clients get replies to their requests even when there are other clients accessing the service. A non-faulty primary assigns sequence numbers using a FIFO discipline. Backups maintain the requests in a FIFO queue and they only stop the view-change timer when the first request in their queue is executed; this prevents faulty primaries from giving preference to some clients while not processing requests from others.

## 5. BFT-PR: BFT WITH PROACTIVE RECOVERY

BFT provides safety and liveness if fewer than  $1/3$  of the replicas fail during the lifetime of the system. These guarantees are insufficient for long-lived systems because the bound is likely to be exceeded in this case. Therefore, we have developed a recovery mechanism for BFT that makes faulty replicas behave correctly again. BFT with recovery, BFT-PR, can tolerate any number of faults provided fewer than  $1/3$  of the replicas become faulty within a small window of vulnerability.

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 mechanism recovers replicas periodically even if there is no reason to suspect that they are faulty.

Section 5.1 describes the additional assumptions required to provide automatic recoveries and Section 5.2 presents the modified algorithm.

### 5.1 Additional Assumptions

To implement recovery, we must mutually authenticate a faulty replica that recovers to the other replicas, and we need a reliable mechanism to trigger periodic recoveries. This can be achieved by involving system administrators in the recovery process, but such an approach is impractical given our goal of

recovering replicas frequently to achieve a small window of vulnerability. To implement automatic recoveries, we need additional assumptions.

*Secure Cryptography.* Each replica has a secure cryptographic coprocessor, for example, a Dallas Semiconductors iButton or the security chip in the motherboard of the IBM PC 300PL. The coprocessor stores the replica's private key, and can sign and decrypt messages without exposing this key. It also contains a counter that never goes backwards. This enables it to append 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. 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. There are extension cards that offer this functionality.

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

Note that all previous proactive security algorithms [Ostrovsky and Yung 1991; Herzberg et al. 1995, 1997; Canetti et al. 1997; Garay et al. 2000] assume the entire program run by a replica is in read-only memory so that it cannot be modified by an attacker, and most also assume that there are authenticated channels between the replicas 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 coprocessors to establish new session keys between the replicas after a recovery.

The only work on proactive security that does not assume authenticated channels is Canetti et al. [1997], but the best that a replica can do when its private key is compromised is alert an administrator. Our secure cryptography assumption enables automatic recovery from most failures, and secure coprocessors with the properties we require are now readily available. We also assume clients have a secure coprocessor; this simplifies the key exchange protocol between clients and replicas but it could be avoided by adding an extra round to this protocol. These assumptions can be relaxed when the goal is to tolerate faults that are not triggered by malicious intelligence.

BFT with proactive recovery needs a stronger synchrony assumption to provide liveness. We assume there is some unknown point in the execution after which either all messages are delivered within some constant time  $\Delta$  (possibly after being retransmitted) or all nonfaulty clients have received replies to their requests. Here,  $\Delta$  is a constant that depends on the timeout values used by the algorithm. This assumption is stronger than the one used so far to allow

recoveries at a fixed rate but it is still likely to hold in real systems with an appropriate choice of  $\Delta$ .

## 5.2 Modified Algorithm

We start by providing an overview of the recovery mechanism. Then we describe it in detail.

**5.2.1 Overview.** BFT uses quorums as a reliable memory to store request ordering information. We must ensure that this memory keeps working in the presence of proactive recoveries. In particular, the proactive recovery mechanism must ensure the following.

Each quorum certificate received by a nonfaulty replica must be backed by a quorum; that is, the states of nonfaulty quorum members must record that a matching message was sent or they must have a later stable checkpoint.

Additionally, the recovery mechanism must ensure that the service state kept by the replica is consistent with the protocol state:

For any nonfaulty replica, the value of the current service state (or any checkpoint) with sequence number  $n$  must be identical to the value obtained by running the requests with sequence numbers between  $h + 1$  and  $n$  in order of increasing number and starting from the stable checkpoint  $h$ . These requests must be committed at the replica.

There are several problems that need to be addressed to ensure that these invariants are preserved when a replica recovers. First, it is necessary to prevent attackers from impersonating replicas that were faulty after they recover. Otherwise, there is no hope of ensuring any of the invariants above. Impersonation can happen if the attacker learns the MAC keys used to authenticate messages but even if messages were signed using the secure cryptographic coprocessor, an attacker would be able to sign bad messages while it controlled a faulty replica. We avoid this problem by changing MAC keys during recoveries and by having replicas and clients reject messages that are authenticated with old keys.

However, changing keys is not sufficient. If a replica collects messages for a certificate over a sufficiently long period of time, it can end up with more than  $f$  messages sent by replicas when they were faulty, which violates the first invariant. We solve this problem by having replicas and clients discard all messages that are not part of a complete certificate when they change keys. To ensure liveness, replicas and clients authenticate the messages that they retransmit with the latest keys. Section 5.2.2 explains how keys are changed.

Since recovery is proactive, a recovering replica may not be faulty and recovery must not cause it to become faulty; otherwise any of the invariants above could be violated. In particular, a nonfaulty replica cannot lose its state and we need to allow it to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery. However, if a recovering replica is actually faulty, the recovery mechanism must ensure that its state is brought to a value that satisfies the invariants above and the replica must be prevented from spreading incorrect

information. The difficulty is that we do not know if the recovering replica is faulty during recovery. We explain how to solve this problem in Section 5.2.3.

**5.2.2 Key Exchanges.** Replicas and clients refresh the session keys used to authenticate messages sent to them by sending NEW-KEY messages periodically (e.g., every minute). The same mechanism is used to establish the initial session keys. The message has the form  $\langle \text{NEW-KEY}, i, \dots, \{k_{j,i}\}_{\epsilon_j}, \dots, t \rangle_{\sigma_i}$ . The message is signed by the secure coprocessor (using the replica's private key) and  $t$  is the value of its counter; the counter is incremented by the coprocessor and appended to the message every time it generates a signature. (This prevents suppress-replay attacks [Gong 1992].) 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, each replica discards its current key for that client, which forces the client to refresh the key.

Let  $t_1$  and  $t_2$  ( $> t_1$ ) be the instants when two consecutive NEW-KEY messages are sent by the same node. We call the interval  $[t_1, t_2]$  a *refreshment epoch*, and its duration,  $t_2 - t_1$ , a *refreshment period*.

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 (with the exception of PRE-PREPARE and PREPARE messages it sent) 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, that is, messages authenticated with keys created in the same refreshment epoch.

**5.2.3 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 its state satisfies the invariants defined before. The protocol goes through the following steps.

**Reboot.** Recovery is proactive—it starts periodically when the watchdog timer goes off. If the recovering replica believes it is in a view  $v$  for which it is the primary, it multicasts a VIEW-CHANGE message for  $v + 1$  just before starting to recover. Any correct replica that receives this message and is in view  $v$  changes to view  $v + 1$  immediately. This improves availability because the backups do not have to wait for their timers to expire before changing to  $v + 1$ . A faulty primary could send such a message and force a view change but this is not a problem because it is always good to replace a faulty primary.

The recovery monitor saves the replica's state (the log, the service state, and checkpoints) 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 can be ensured by storing their digest in the read-only memory and by having the recovery monitor check this digest. If the copy of the code stored by the replica is corrupt, the recovery monitor can fetch the correct code from the other replicas. Alternatively, the entire code can be stored in a read-only medium; this is feasible because there are several disks that can be write protected by physically closing a jumper switch (e.g., the Seagate Cheetah 18LP). Rebooting restores the operating system data structures to a correct state and removes any Trojan horses left by an attacker.

After this point, the recovering 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 recovery protocol solves these problems as described next.

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 log entries or checkpoints with greater sequence numbers. This bounds the sequence numbers of any incorrect messages sent by the replica while ensuring that no state is discarded when the replica is not faulty.

Estimation works as follows:  $i$  multicasts a  $\langle \text{QUERY-STABLE}, i \rangle_{\alpha_i}$  message to the other replicas. When replica  $j$  receives this message, it replies  $\langle \text{REPLY-STABLE}, c, p, i \rangle_{\mu_{ji}}$ , where  $c$  and  $p$  are the sequence numbers of the last checkpoint and the last request prepared at  $j$ , respectively. Replica  $i$  keeps retransmitting the query message and processing replies; it keeps the minimum value of  $c$  and the maximum value of  $p$  it receives from each replica. It also keeps its own values of  $c$  and  $p$ . During estimation  $i$  does not handle any other protocol messages except NEW-KEY and REPLY-STABLE.

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 one replica  $j$  that satisfies two conditions:  $2f$  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 the sequence number of any stable checkpoint  $i$  may have when it is not faulty so that it will not discard log entries in this case. This is ensured because if a checkpoint is stable, it will have been created by at least  $f + 1$  nonfaulty 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 nonfaulty replica since at least

one nonfaulty 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  $2f + 1$  nonfaulty replicas and they only propose a value of  $c$  if the corresponding request committed; this implies that it prepared at at least  $f + 1$  correct replicas. Therefore  $i$  can always base its choice of  $c_M$  on the set of messages sent by correct replicas.

After this point  $i$  participates in the protocol as if it were not recovering but it will not send any messages with sequence numbers above  $H_M$  until it has a correct stable checkpoint with sequence number greater than or equal to  $H_M$ . This ensures a bound  $H_M$  on the sequence number of any bad messages  $i$  may send based on corrupt state.

*Send Recovery Request.* Next  $i$  multicasts a recovery request to the other replicas with the form:  $\langle \text{REQUEST}, \langle \text{RECOVERY}, H_M \rangle, t, i \rangle_{\sigma_i}$ . This message is produced by the cryptographic coprocessor and  $t$  is the coprocessor'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 nonfaulty replicas are kept busy executing recovery requests.

The recovery request is treated as 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 6.2.2) and determine that it reflects the execution of a new recovery request. This is important because these keys may be 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; that is,  $H_R = \lfloor n_R/K \rfloor \times K + L$ .

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,  $H = \max(H_M, H_R)$ . The replica also computes a valid view: it retains its current view  $v_r$  if there are  $f + 1$  replies to the recovery request with views greater than or equal to  $v_r$ , else it changes to the median of the views in the replies. The replica also retains its view if it changed to that view after recovery started. If the replica changes its view, it sends a VIEW-CHANGE message for  $v_m$  and it waits for a correct NEW-VIEW message and a matching set of VIEW-CHANGE messages before becoming active in  $v_m$ .

The mechanism to compute a valid view ensures that nonfaulty replicas never change to a view with a number smaller than their last active view. If the recovering replica is correct and has an active view with number  $v_r$ , there is a quorum of replicas with view numbers greater than or equal to  $v_r$ . Therefore the recovery request will not prepare at any correct replica with a view number smaller than  $v_r$ . Additionally, the median of the view numbers in replies to the recovery request will be greater than or equal to the view number in a reply



from a correct replica. Therefore it will be greater than or equal to  $v_r$ . Changing to the median  $v_m$  of the view numbers in the replies is also safe because at least one correct replica executed the recovery request at a view number greater than or equal to  $v_m$ .

*Check and Fetch State.* While  $i$  is recovering, it uses the state transfer mechanism discussed in Section 6.2.3 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 it has a stable checkpoint with sequence number greater than or equal to  $H$ . If clients aren't using the system this could delay recovery, since request number  $H$  needs to execute for recovery to complete. However, this is easy to fix. While a recovery is occurring, the primary sends PRE-PREPARES for null requests.

Our protocol has the nice property that any replica knows that  $i$  has completed its recovery when checkpoint  $H$  is stable and they have received a CHECKPOINT message from  $i$ . 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.

**5.2.4 Improved Service Properties.** BFT-PR ensures safety and liveness for an execution  $\tau$  provided at most  $f$  replicas fail within any time interval of size  $T_v = 2T_k + T_r$ . Here,  $T_v$  is the window of vulnerability,  $T_k$  is the maximum key refreshment period in  $\tau$  for a nonfaulty node, and  $T_r$  is the maximum time between when a replica fails and when it recovers from that fault in  $\tau$ . Note that the values of  $T_k$  and  $T_r$  are characteristic of each execution  $\tau$  and unknown to the algorithm.

It is necessary to set the window of vulnerability to a value greater than or equal to  $2T_k + T_r$  to ensure that correct nodes do not collect certificates with more than  $f$  bad messages. There would be no hope of preserving the invariants listed in Section 5.2.1 with a smaller window. The session key refreshment mechanism ensures that nonfaulty nodes only accept certificates with messages generated within an interval of size at most  $2T_k$ .<sup>1</sup> In addition, bounding the number of replicas that can fail within an interval of size  $T + T_r$  (for any  $T$ ) ensures that there are never more than  $f$  faulty replicas within any interval of size at most  $T$ . Therefore, any certificate collected by a correct node will include at most  $f$  messages sent by replicas when they were faulty.

Next we argue that the recovery mechanism preserves the invariants listed in Section 5.2.1. We designed the recovery mechanism to ensure that nonfaulty replicas do not lose their state when they recover. Therefore the invariants are preserved in this case. The invariants are also preserved when the recovering replica is faulty. This is true because other correct replicas do not accept bad messages sent by the recovering replica with sequence number greater than the recovery point. In addition, the replica has a correct log and a correct stable checkpoint with sequence number equal to the recovery point by the

<sup>1</sup>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.

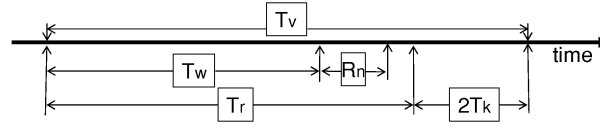


Fig. 5. Relationship between the window of vulnerability  $T_v$  and other time intervals.

end of recovery. This ensures that the replica has a stable checkpoint with sequence number greater than any message it sent before and during recovery that may have been accepted as part of a certificate by another replica or client.

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 trade-off 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 8.2.3 shows that these timeouts can be quite small with low performance degradation.

The period between key changes  $T_k$  can be small without having a significant impact on performance (e.g., 15 seconds). But  $T_k$  should be substantially larger than three message delays under normal load conditions to provide liveness.

The value of  $T_w$  should be set based on  $R_n$ , the time it takes to recover a nonfaulty 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). Figure 5 shows the relationship between the various time intervals.

The results in Section 8.2.3 indicate that  $R_n$  is dominated by the time to reboot and check the correctness of the replica's copy of the service state. Since a replica that is not faulty checks its state without placing much load on the network or any other replica, we expect the time to recover  $f$  replicas in parallel and the time to recover a replica under benign overload conditions to be close to  $R_n$ ; thus we can set  $s$  close to 1.

We cannot guarantee any bound on  $T_v$  under a denial-of-service attack but it is possible for replicas to time recoveries and alert an administrator if they take longer than some constant times  $R_n$ . The administrator can then take action to allow the recovery to terminate. For example, if replicas are connected by a private network, they may stop processing incoming requests and use the private network to complete recovery. This will interrupt service until recovery completes but it does not give any advantage to the attacker; if the attacker can prevent recovery from completing, it can also prevent requests from executing. It may be possible to automate this response.

Replicas should also log information about recoveries, including whether there was a fault at a recovering node, and how long the recovery took, since this information is useful to strengthen the system against future attacks.

## 6. IMPLEMENTATION TECHNIQUES

This section describes protocol optimizations and checkpoint management.

### 6.1 Optimizations

This section describes optimizations that improve the performance during normal case operation while preserving the safety and liveness properties. The most important optimization was already described: BFT uses MACs based on symmetric cryptography to authenticate messages instead of public key signatures. Since MACs can be computed three orders of magnitude faster, this optimization is quite effective.

*Digest Replies.* The second optimization reduces network bandwidth consumption and CPU overhead significantly when operations have large results. A client request designates a replica to send the result. This replica may be chosen randomly or using some other load balancing scheme. After the designated replica executes the request, it sends back a reply containing the result. The other replicas send back replies containing only the digest of the result. The client collects at least  $f + 1$  replies (including the one with the result) and uses the digests to check the correctness of the result. If the client does not receive a correct result from the designated replica, it retransmits the request (as usual) requesting all replicas to send replies with the result.

*Tentative Execution.* The third optimization reduces the number of message delays for an operation invocation from five to four. Replicas execute requests tentatively as soon as: they have a prepared certificate for the request, their state reflects the execution of all requests with lower sequence number, and these requests have committed. After executing the request, the replicas send tentative replies to the client. Since replies are tentative, the client must wait for a quorum certificate with replies with the same result. This ensures that the request is prepared by a quorum and, therefore, it is guaranteed to commit eventually at nonfaulty replicas. If the client's retransmission timer expires before it receives these replies, the client retransmits the request and waits for nontentative replies.

A request that has executed tentatively may abort if there is a view change. In this case, the replica reverts its state to the checkpoint in the NEW-VIEW message or to its last checkpointed state (depending on which one has the higher sequence number).

It is possible to take advantage of tentative execution to eliminate COMMIT messages: they can be piggybacked in the next PRE-PREPARE or PREPARE message sent by a replica. Since clients receive replies after a request prepares, piggybacking COMMITs does not increase latency and it reduces load both on the network and on the replicas' CPUs.

*Read-Only Operations.* This optimization improves the performance of read-only operations, which do not modify the service state. A client multicasts a read-only request to all replicas. The replicas execute the request immediately after checking that it is properly authenticated, the client has access, and the request is in fact read-only. A replica sends back a reply only after all requests it executed before the read-only request have committed. The client waits for a quorum certificate with replies with the same result. It may be unable to collect this certificate if there are concurrent writes to data that affect the result. In this case, it retransmits the request as a regular read-write request after its retransmission timer expires.

The read-only optimization preserves linearizability provided clients obtain a quorum certificate with replies not only for read-only operations but also for any read-write operation. This optimization reduces latency to a single round trip for most read-only requests.

*Request Batching.* Batching reduces protocol overhead under load by assigning a single sequence number to a batch of requests and by starting a single instance of the protocol for the batch. We use a sliding-window mechanism to bound the number of protocol instances that can run in parallel. Let  $e$  be the sequence number of the last batch of requests executed by the primary and let  $p$  be the sequence number of the last PRE-PREPARE sent by the primary. When the primary receives a request, it starts the protocol immediately unless  $p \geq e + W$ , where  $W$  is the window size. In the latter case, it queues the request. When requests execute, the window slides forward allowing queued requests to be processed. Then the primary picks the first requests from the queue such that the sum of their sizes is below a constant bound, it assigns them a sequence number, and it sends them in a single PRE-PREPARE message. The protocol proceeds exactly as it did for a single request except that replicas execute the batch of requests (in the order in which they were added to the PRE-PREPARE message) and they send back separate replies for each request.

## 6.2 Checkpoint Management

BFT's garbage collection mechanism (see Section 4.4) takes logical snapshots of the service state called *checkpoints*. These snapshots are used to replace messages that have been garbage collected from the log. This section describes a technique to manage checkpoints. It starts by describing checkpoint creation, computation of checkpoint digests, and the data structures used to record checkpoint information. Then, it describes a state transfer mechanism that is used to bring replicas up to date when some of the messages that they are missing were garbage collected. It ends with an explanation of the mechanism used to check the correctness of a replica's state during recovery.

**6.2.1 Data Structures.** We use hierarchical state partitions to reduce the cost of computing checkpoint digests and the amount of information transferred to bring replicas up to date. The root partition corresponds to the entire service state and each nonleaf partition is divided into  $s$  equal-sized, contiguous

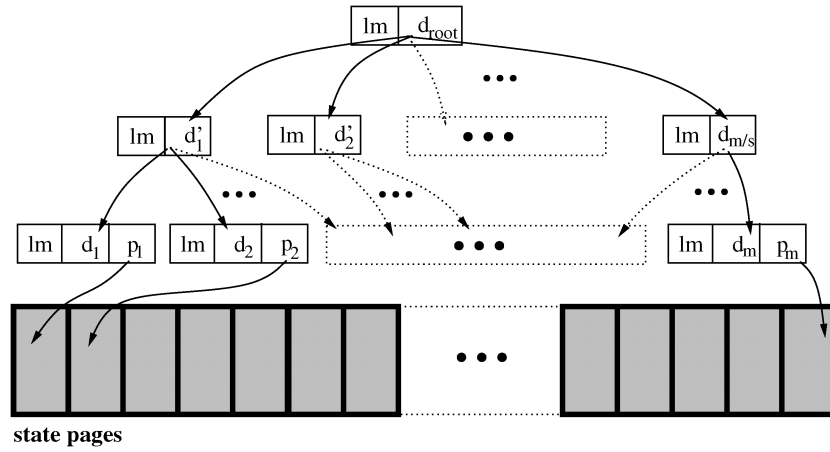


Fig. 6. Partition tree.

subpartitions. Figure 6 depicts a partition tree with three levels. We call the leaf partitions *pages* and the interior ones *metadata*. For example, the experiments described in Section 8 were run with a hierarchy with four levels,  $s$  equal to 256, and 4-KB 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. Checkpoints are taken immediately after tentatively executing a request batch with sequence number divisible by the checkpoint period  $K$  (but the corresponding CHECKPOINT messages are sent only after the batch commits).

The tree for a checkpoint stores a tuple  $\langle lm, d \rangle$  for each metadata 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 epoch where the partition was modified,  $d$  is the digest of the partition, and  $p$  is the value of the page. Partition digests are important. Replicas use the digest of the root partition during view changes to agree on a start state for request processing in the new view without transferring a large amount of data. They are also used to reduce the amount of data sent during state transfer.

The digests are computed efficiently as follows. A page digest is obtained by applying a cryptographic hash function (currently MD5 [Rivest 1992]) to the string obtained by concatenating the index of the page within the state, its value of  $lm$ , and  $p$ . A metadata digest is obtained by applying the hash function 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 subpartitions. Thus, we apply AdHash [Bellare and Micciancio 1997] at each metadata 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. It is inspired by Merkle trees [Merkle 1987].

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.

**6.2.2 State Transfer.** A replica initiates a state transfer when it learns about a stable checkpoint with sequence number greater than the high water mark in its log. It uses the state transfer mechanism to fetch modifications to the service state that it is missing. The replica may learn about such a checkpoint by receiving CHECKPOINT messages or as the result of a view change.

It is important for the state transfer mechanism to be efficient because it is used to bring a replica up to date during recovery and we perform proactive recoveries frequently. The key issues to achieving efficiency are reducing the amount of information transferred and reducing the burden imposed on other replicas. The strategy to fetch state efficiently is to recurse down the partition hierarchy to determine which partitions are out of date. This reduces the amount of information about (both nonleaf and leaf) partitions that needs to be fetched.

The state transfer mechanism must also ensure that the transferred state is correct even when some replicas are faulty or the state is modified concurrently. The idea is that the digest of a partition commits the values of all its subpartitions for a particular sequence number. A replica starts a state transfer by obtaining a weak certificate with the digest of the root partition at some checkpoint  $c$ . Then it uses this digest to verify the correctness of the subpartitions it fetches. The replica does not need a weak certificate for the subpartitions unless the value of a subpartition at checkpoint  $c$  has been discarded. The next paragraphs describe the state transfer mechanism in more detail.

A replica  $i$  multicasts  $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$  to all other 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 nil 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 number. The value of  $c$  is not nil when  $i$  knows the correct digest of the partition at checkpoint  $c$ ; for example, 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 subpartition 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 a certificate or even authentication. This reduces the burden imposed on other replicas and it is important to provide liveness in view changes when the

start state for request processing in the new view is held by a single correct replica.

Replicas other than the designated replier 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 a weak certificate with equally fresh responses with the same subpartition values for the same sequence number  $c'$  (greater than  $lc$  and  $c$ ). Then it compares its digests for each subpartition of  $(l, x)$  with those in the fetched information; it multicasts a `FETCH` message for subpartitions where there is a difference, and sets the value in  $LC$  to  $c$  (or  $c'$ ) for the subpartitions that are up to date. Since  $i$  learns the correct digest of each subpartition at checkpoint  $c$  (or  $c'$ ), it can use the optimized protocol to fetch them using these digests to check if they are correct.

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 subpartitions, 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  $LC$ . 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  $LC$  for its subpartitions, and  $c$  is greater than or equal to the maximum of the last modification sequence numbers in its subpartitions. 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; for example, each time it fetches a missing partition, it receives information about a yet later modification. If the service operations change data faster than they can be transferred, an out-of-date replica may never catch up. The state transfer mechanism described can transfer data fast enough that this is unlikely to be a problem for most services. The transfer rate could be improved by fetching pages in parallel from different replicas but this is not currently implemented. Furthermore, if the

replica fetching the state is ever actually needed (because others have failed), the system will wait for it to catch up.

**6.2.3 State Checking.** It is necessary to ensure that a replica's state is both correct and up to date after recovery. This is done by using the state transfer mechanism to fetch out-of-date pages and to obtain the digests of up-to-date partitions; the recovering replica uses these digests to check if its copies of the partitions are correct.

The recovering replica starts by computing the partition digests for all meta-data assuming that the digests for the pages match the values it stores. Then, it initiates a state transfer as described above except that the value of *lc* in the first `FETCH` message for each metadata partition is set to  $-1$ . This ensures that the `META-DATA` replies include digests for all subpartitions.

The replica processes replies to `FETCH` messages as described before but, rather than ignoring up-to-date partitions, it checks if the partition digests match the ones it has recorded in the partition tree. If they do not, the partition is queued for fetching as if it were out of date; otherwise, the partition is queued for checking.

Partition checking is overlapped with the time spent waiting for fetch replies. A replica checks a partition by computing the digests for each of the partition's pages and by comparing those digests with the ones in the partition tree. Those pages whose digests do not match are queued for fetching.

## 7. THE BFT LIBRARY

The algorithm has been implemented as a generic program library with a simple interface. The library can be used to provide Byzantine-fault-tolerant versions of different services. Section 7.1 describes the library's implementation and Section 7.2 presents its interface. We used the library to implement a Byzantine-fault-tolerant NFS file system, which is described in Section 7.3.

### 7.1 Implementation

The library uses a connectionless model of communication: point-to-point communication between nodes is implemented using UDP [Postel 1980], and multicast to the group of replicas is implemented using UDP over IP multicast [Deering and Cheriton 1990]. There is a single IP multicast group for each service, which contains all the replicas. Clients are not members of this multicast group (unless they are also replicas).

The library is implemented in C++. We use an event-driven implementation with a structure very similar to the I/O automaton code in the formalization of the algorithm in the Appendix. Replicas and clients are single threaded and their code is structured as a set of event handlers. This set contains a handler for each message type and a handler for each timer. Each handler corresponds to an input action in the formalization and there are also methods that correspond to the internal actions. The similarity between the code and the formalization is intentional and it was important: it helped identify several errors in the code and omissions in the formalization.



```

Client:
int Byz_init_client(char *conf);
int Byz_invoke(Byz_req *req, Byz_rep *rep, bool ro);

Server:
int Byz_init_replica(char *conf, char *mem, int size, proc exec, proc nondet);
void Byz_modify(char *mod, int size);

Server upcalls:
int execute(Byz_req *req, Byz_rep *rep, Byz_buffer *ndet, int cid, bool ro);

int nondet(Seqno seqno, Byz_req *req, Byz_buffer *ndet);

```

Fig. 7. The replication library API.

The event handling loop works as follows. Replicas and clients wait in a select call for a message to arrive or for a timer deadline to be reached and then they call the appropriate handler. The handler performs computations similar to the corresponding action in the formalization and then it invokes any methods corresponding to internal actions whose preconditions become true. The handlers never block waiting for messages.

We use the SFS [Mazières et al. 1999] implementation of a Rabin–Williams public key cryptosystem with a 1,024-bit modulus to establish 128-bit session keys. All messages are then authenticated using message authentication codes computed using these keys and UMAC32 [Black et al. 1999]. Message digests are computed using MD5 [Rivest 1992].

The implementation of public key cryptography signs and encrypts messages as described in Bellare and Rogaway [1996] and [1995], respectively. These techniques are provably secure in the random oracle model [Bellare and Rogaway 1995]. In particular, signatures are nonexistentially forgeable even with an adaptive chosen message attack. UMAC32 is also provably secure in the random oracle model. MD5 should still provide adequate security and it can be replaced easily by another hash function (e.g., SHA-1 [SHA1 1994]) at the expense of some performance degradation.

The message formats are designed such that the MACs are computed only over a fixed-size header. This has the advantage of making the cost of authenticator computation, which grows linearly with the number of replicas, independent of the payload size (e.g., independent of the operation argument size in requests and the size of the batch in PRE-PREPARES).

## 7.2 Interface

We implemented the algorithm as a library with a very simple interface (see Figure 7). Some components of the library run on clients and others at the replicas.

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

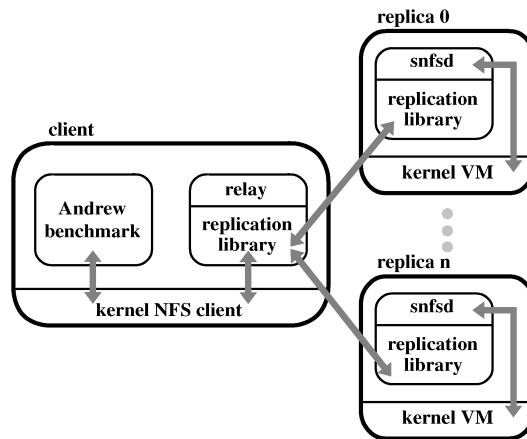


Fig. 8. BFS: replicated file system architecture.

and returns the result when enough replicas have responded. The library also provides a split interface (not shown in the figure) with separate send and receive calls to invoke requests.

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 service state is stored, a procedure to execute requests, and a procedure to compute nondeterministic choices. When our system needs to execute an operation, it does an upcall to the *execute* procedure. The arguments to this procedure include a buffer with the requested operation and its arguments *req*, and a buffer to fill with the operation result *rep*. The *execute* procedure carries out the operation as specified for the service, using the service state. As the service performs the operation, each time it is about to modify the service state, it calls the *modify* procedure to inform the library of the locations about to be modified. This call allows us to maintain checkpoints and compute digests efficiently as described in Section 6.2.2.

Additionally, the *execute* procedure takes as arguments the identifier of the client who requested the operation and a Boolean flag indicating whether the request was processed with the read-only optimization. The service code uses this information to perform access control and to reject operations that modify the state but were flagged read-only by faulty clients. When the primary receives a request, it selects any nondeterministic input to the requested operation (e.g., a timestamp) by making an upcall to the *nondet* procedure. The BFT library ensures that replicas agree on this nondeterministic input and it is passed as an argument to the *execute* upcall [Castro 2001].

### 7.3 BFS: A Byzantine-Fault-Tolerant File System

We implemented BFS, a Byzantine-fault-tolerant NFS [Sandberg et al. 1985] service, using the replication library. BFS implements version 2 of the NFS protocol. Figure 8 shows the architecture of BFS. A file system exported by the fault-tolerant NFS service is mounted on the client machine like any regular

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

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

We implemented *snfsd* using a fixed-size memory-mapped file. All the file system data structures (e.g., inodes, blocks, and their free lists) are in the mapped file. We rely on the operating system to manage the cache of memory-mapped file pages and to write modified pages to disk asynchronously. The current implementation uses 4-KB blocks and inodes contain the NFS status information plus 256 bytes of data, which are used to store directory entries in directories, pointers to blocks in files, and text in symbolic links. Directories and files may also use indirect blocks in a way similar to UNIX.

Our implementation ensures that all state machine replicas start in the same initial state and are deterministic, which are necessary conditions for the correctness of a service implemented using our protocol. The primary proposes the values for time-last-modified and time-last-accessed, and replicas select the larger of the proposed value and one greater than the maximum of all values selected for earlier requests. The primary selects these values by executing the upcall to compute nondeterministic choices, which simply returns the result of *gettimeofday* in this case.

We do not require synchronous writes to implement NFS V2 protocol semantics because BFS achieves stability of modified data and metadata through replication as was done in Harp [Liskov et al. 1991]. If power failures are likely to affect all replicas, each replica should have an uninterruptible power supply (UPS). The UPS will allow enough time for a replica to write its state to disk in the event of a power failure as was done in Harp [Liskov et al. 1991].

## 8. PERFORMANCE EVALUATION

The BFT library can be used to implement Byzantine-fault-tolerant systems but these systems will not be used in practice unless they perform well. This section presents results of experiments to evaluate the performance of these systems.

We ran several benchmarks to measure the performance of BFS, our Byzantine-fault-tolerant NFS. The results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol, which are used daily by many users and are not replicated. Additionally, we ran microbenchmarks to evaluate the performance of the replication library in a service-independent way. We presented a detailed analytic performance model and experiments to evaluate the impact of each optimization in Castro [2001].

## 8.1 Microbenchmarks

This section presents results of microbenchmarks. The experiments were performed using the setup in Section 8.1.1. Sections 8.1.2 and 8.1.3 describe experiments to measure the latency and throughput of a simple replicated service with four replicas. We investigate the impact on performance as the number of replicas increases in Section 8.1.4. The experiments in these sections evaluate performance without checkpoint management, view changes, or recovery. In Sections 8.1.5 and 8.1.6, we analyze the performance overhead introduced by checkpoint management, and view changes. Performance with recoveries is studied in Section 8.2.3.

**8.1.1 *Experimental Setup.*** The experiments ran on nine Dell Precision 410 workstations with a single Pentium III processor, 512 MB of memory, and a Quantum Atlas 10 K 18 WLS disk. All machines ran Linux 2.2.16-3 compiled without SMP support. The processor clock speed was 600 MHz in seven machines and 700 MHz in the other two. All experiments ran on the slower machines except where noted. The 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. All experiments ran on an isolated network.

The experiments compare the performance of two implementations of a simple service: one implementation, BFT, is replicated using the BFT library and the other, NO-REP, is not replicated and uses UDP directly for communication between the clients and the server without authentication. The simple service is really the skeleton of a real service: it has no state and the service operations receive arguments from the clients and return (zero-filled) results but they perform no computation. We performed experiments with different argument and result sizes for both read-only and read-write operations. It is important to note that this is a worst-case comparison; in real services, computation or I/O at the clients and servers would reduce the slowdown introduced by the BFT library (as shown in Section 8.2).

The library was configured as follows: the period between checkpoints was 128 sequence numbers, the size of the log was 256 sequence numbers, and the window size for request batching was 1.

**8.1.2 *Latency.*** We measured 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 3% of the reported values. Figure 9 shows the latency to invoke the replicated service as the size of the operation result increases while keeping the argument size fixed at 8-B. It has one graph with elapsed times and another with the slowdown of BFT relative to NO-REP.

Figure 10 shows the latency to invoke the replicated service as the size of the operation argument increases while keeping the result size fixed at 8 bytes. The two figures have results for both read-write and read-only operations.

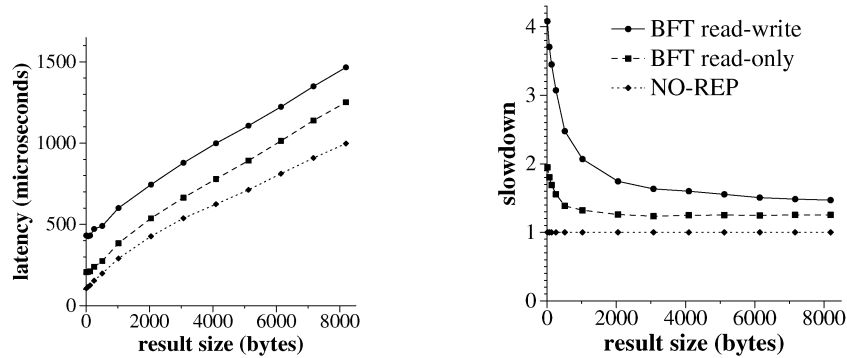


Fig. 9. Latency with varying result sizes: absolute times and slowdown relative to NO-REP.

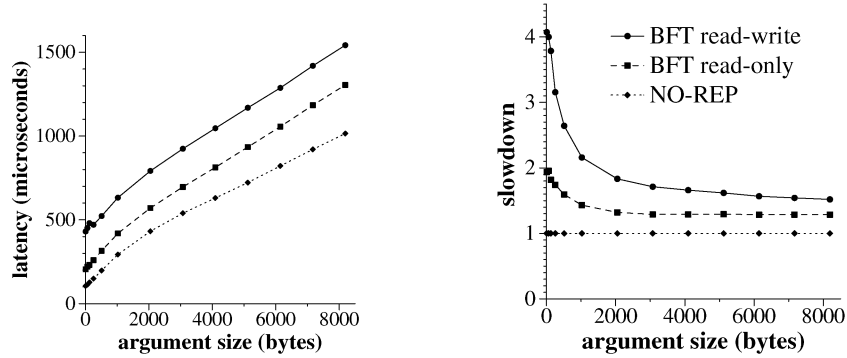


Fig. 10. Latency with varying argument sizes: absolute times and slowdown relative to NO-REP.

The library introduces a significant slowdown relative to NO-REP but the slowdown decreases quickly as the operation argument or result sizes increase. For example, the slowdown for the read-write operation decreases from 4.08 with 8-B results to 1.47 with 8-KB results, and it decreases from 1.95 to 1.25 with the read-only optimization. The two major sources of overhead are digest computation (of requests and replies) and the additional communication due to the replication protocol. The cost of MAC computation is negligible.

The latency increases because the communication time to send the reply (or request) and the time to digest the reply (or request) grow with the result (or argument) size. In our experimental setup, the communication time increases 91 ns/byte and the digest computation time increases 24 ns/byte. Since the latency of NO-REP also increases 91 ns/byte, the slowdown decreases as the result or argument size increases until an asymptote of  $(91 + 24)/91 = 1.26$ .

The read-only optimization is very effective at reducing the slowdown introduced by the BFT library. It improves performance by eliminating the time to prepare the requests. This time does not change as the argument or result size increases. Therefore, the speedup afforded by the read-only optimization decreases to zero as the argument or result size increases. For example, it reduces latency by 52% with 8-B arguments but only by 15% for 8-KB arguments.

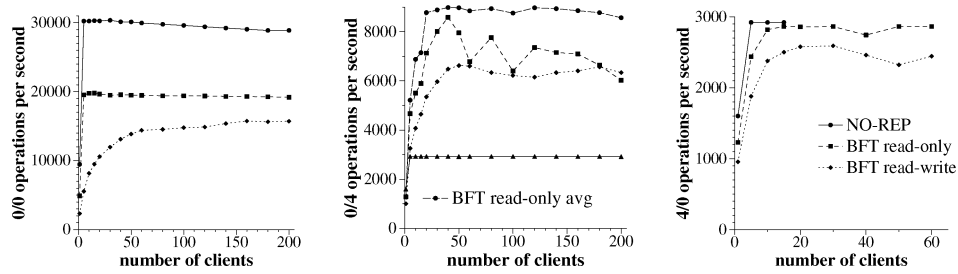


Fig. 11. Throughput for operations 0/0, 0/4, and 4/0.

**8.1.3 Throughput.** This section reports the result of experiments to measure the throughput of BFT and NO-REP as a function of the number of clients accessing the simple service. The client processes were evenly distributed over five client machines.<sup>2</sup> We measured throughput for operations with different argument and result sizes. Each operation type is denoted by  $a/b$ , where  $a$  and  $b$  are the sizes of the argument and result in KB.

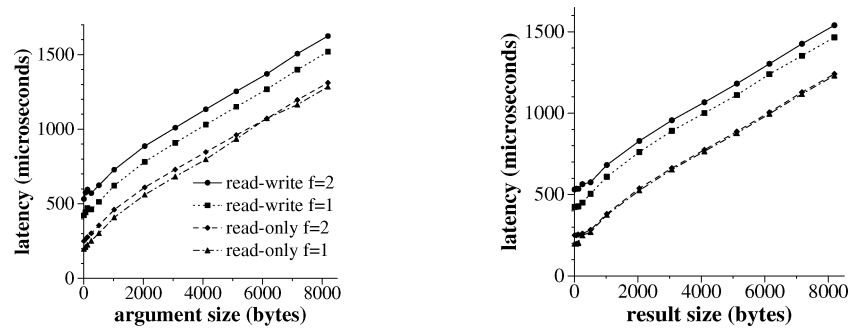
The experiment ran as follows: all client processes started invoking operations almost simultaneously; each client process executed  $3K$  operations (where  $K$  was a large number) and measured the time to execute the middle  $K$  operations. The throughput was computed as  $K$  multiplied by the number of client processes and divided by the maximum time (taken over all clients) to complete the  $K$  operations. This methodology provides a conservative throughput measurement: it accounts for cases where clients are not treated fairly and take longer to complete the  $K$  iterations. Each throughput value reported is the average of at least three independent runs.

Figure 11 shows throughput results for operations 0/0, 0/4, and 4/0. The standard deviation was below 7% of the reported values except for read-only operation 0/4 (where it was as high as 18%).

The bottleneck in operation 0/0 is the server's CPU. BFT has lower throughput than NO-REP due to extra messages and cryptographic operations that increase the CPU load. BFT's throughput is 52% lower for read-write operations and 35% lower for read-only operations. The read-only optimization improves throughput by eliminating the cost of preparing the batch of requests. The throughput of the read-write operation improves as the number of clients increases because the cost of preparing the batch of requests is amortized over the size of the batch. The throughput saturates because we bound the number of requests in a batch as a defense against denial-of-service attacks.

BFT has better throughput than NO-REP for operation 0/4. The bottleneck for NO-REP is the link bandwidth (12 MB/s); it executes approximately 3,000 operations per second. BFT achieves better throughput because of the digest-replies optimization: clients obtain the replies with the 4-KB result in parallel from different replicas. BFT achieves a maximum throughput of 6,625 operations per second (26 MB/s) for the read-write operation and 8,698 operations

<sup>2</sup>Two client machines had 700-MHz PIIIs but were otherwise identical to the other machines.

Fig. 12. Latency with varying argument and result sizes with  $f = 2$ .

per second (34 MB/s) with the read-only optimization. The bottleneck for BFT is the replicas' CPU.

The throughput for operation 0/4 with the read-only optimization is very unstable because the system is not fair to all clients; there is a large variance in the maximum time to complete the  $K$  operations. The average time to compute these operations remains stable, as shown by the throughput values labeled “avg,” which are computed using this time.

The bottleneck in operation 4/0 for both NO-REP and BFT is the time to get the requests through the network. Since the link bandwidth is 12 MB/s, the maximum throughput achievable is 3,000 operations per second. NO-REP achieves a maximum throughput of 2,921 operations per second and BFT achieves 2,591 for read-write operations (11% less than NO-REP) and 2,865 with the read-only optimization (2% less than NO-REP). There are no points with more than 15 clients for NO-REP because of lost request messages; NO-REP uses UDP directly and does not retransmit requests.

**8.1.4 Configurations with More Replicas.** The experiments in the previous sections ran in a configuration with four replicas, which can tolerate one fault. We believe this level of reliability will be sufficient for most applications. But some applications will have more stringent reliability requirements and will need to run in configurations with more replicas. Therefore, it is important to understand how the performance of a service implemented with the BFT library is affected when the number of replicas increases. Figure 12 compares the latency to invoke the replicated service with four replicas ( $f = 1$ ) and seven replicas ( $f = 2$ ): the first graph shows latency as a function of argument size, and the second shows latency as a function of the result size. The standard deviation was always below 2% of the reported value. In both configurations, all the replicas had a 600-MHz Pentium III processor and the client had a 700-MHz Pentium III processor.

The results show that the slowdown caused by increasing the number of replicas to seven is low. The maximum slowdown is 30% for the read-write operation and 26% for the read-only operation. Furthermore, the slowdown decreases quickly as the argument or result size increases. For example, with an argument size of 8 KB, the slowdown is only 7% for the read-write operation and

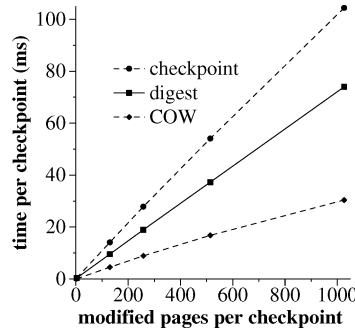


Fig. 13. Checkpoint cost with a varying number of modified pages per checkpoint epoch.

2% with the read-only optimization. The slowdown decreases as the argument size increases because the overhead introduced by adding replicas is independent of this size. The digest replies optimization makes the overhead introduced by adding replicas independent of the result size, which explains why the slowdown also decreases as the result size increases.

**8.1.5 Checkpoint Management.** The experiments in the previous sections used a simple service that had no state. The only checkpoint management overhead in those experiments was due to storing the last replies to read-write operations sent to each client. This section analyzes the performance overhead introduced by checkpoint management using a modified version of the simple service that adds state. The state in the new service is a persistent array of contiguous pages that is implemented by the replicas using a memory-mapped file with 256 MB. The service operations can read or write these pages. The experiments ran with one client and four replicas. This section presents results of experiments to measure both the time to create checkpoints and the time for state transfer to bring replicas up to date.

**Checkpoint Creation.** The checkpoints are created using the technique described in Section 6.2. In our experimental setup, the state partition tree has four levels, each internal node has 256 children, and the pages (i.e., the leaves of the tree) have 4 KB. The requests that execute between two checkpoints are said to be in the same checkpoint epoch.

The cost of checkpoint creation has two components: the time to perform copy-on-write (COW) and the time to compute the checkpoint digest. Figure 13 shows the values we measured for these times with a varying number of modified pages per checkpoint epoch. The results show that both the time to perform copy-on-write and the time to compute digests grow linearly with the number of distinct pages modified during a checkpoint epoch: it costs approximately 72  $\mu$ s to digest each page and 29  $\mu$ s to copy a page.

The cost of checkpoint creation can represent a substantial fraction of the average cost to run an operation when the rate of change is high. It is possible to improve performance by computing checkpoint digests lazily. The protocol can be modified not to send checkpoint digests in CHECKPOINT messages. Thus checkpoint digests would need to be computed only before a view change or a



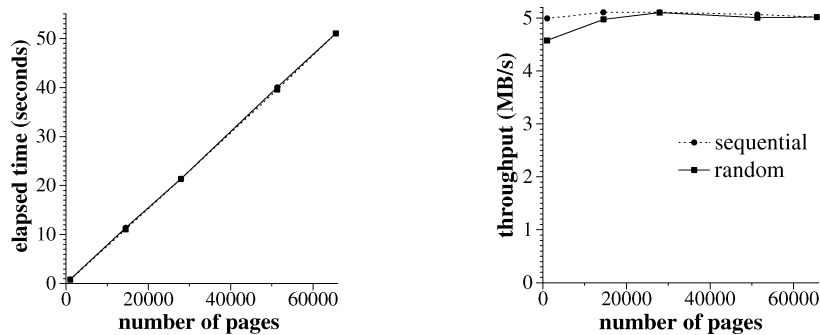


Fig. 14. State transfer latency and throughput.

state transfer. This has the potential of substantially reducing the overhead during the normal case at the expense of potentially slower view changes and state transfers.

*State Transfer.* We also ran experiments to measure the time to complete a state transfer. A client invoked operations that modified a certain number of pages  $m$ . Then the client was stopped and one of the backups was restarted from its initial state. We measured the time to complete the state transfer to bring that backup up to date in an idle system. The experiment was run for several values of  $m$  both with randomly chosen pages and with pages chosen sequentially. Figure 14 shows the elapsed time to complete the state transfer and its throughput.

The results show that the time to complete the state transfer is proportional to the number of pages that are out of date. The throughput is approximately equal to 5 MB/s except that it is 4.5 MB/s when fetching 1,000 random pages. The throughput is lower with random pages because it is necessary to fetch more metadata information but this additional overhead is dwarfed by the time to fetch a large number of pages. The time to complete the state transfer is dominated by the time to fetch data pages and the time to compute their digests to check correctness.

If the rate of modifications to the state is greater than the state transfer throughput, an out-of-date replica may be unable to catch up. This problem may decrease availability: if there is a fault, the system will stop processing client requests until the out-of-date replica can complete the state transfer. There are several ways to ameliorate this problem. The throughput of state transfer can be improved by fetching pages in parallel from all replicas; this should improve throughput to the link bandwidth (12 MB/s). In addition, the replicas can give priority to handling of `FETCH` requests.

**8.1.6 View Changes.** The experiments described so far analyze the performance of the system when there are no faults. This section studies the performance of the view-change protocol. It measures the time from the moment a replica sends a `VIEW-CHANGE` message until it is ready to start processing requests in the new view. This time includes not only the time to receive and process the

Table I. Average View Change Time with Varying Write Percentage

	idle	10%	50%
View-change time ( $\mu$ s)	575	4162	7005

NEW-VIEW message but also the time to obtain any missing requests and, if necessary, the checkpoint chosen as the starting point for request processing in the new view.

We measured the time to complete the view change protocol using the simple service with 256 MB of state, 4-KB pages, and four replicas. There was a single client that invoked two types of operations: a read-only operation that returned the value of a page, and a write operation that wrote a page to the state. The client chose the operation type and the page randomly. View changes were triggered by a separate process that multicast special messages that caused all replicas to move to the next view at approximately the same time.

Table I shows the time to complete a view change for an idle system, and when the client executes write operations with 10 and 50% probability. For each experiment, we timed 128 view changes at each replica and present the average value taken over all replicas.

Replicas never pre-prepare any request in the idle system. Therefore this case represents the minimum time to complete a view-change. This time is only 34% greater than the latency of operation 0/0 on the simple service. The view change time increases when replicas process client requests because VIEW-CHANGE messages include information about messages sent by the replica in previous views.

The increase in the view-change time from 10 to 50% writes is mostly due to one view change that took 607 ms to complete because the replica was out of date and had to fetch a missing checkpoint before it could start processing requests in the new view; the probability of this type of event increases with the rate of modifications to the state.

Since the cost of the view-change protocol in our library is small, we can set the view-change timeout to a small value (e.g., less than a second) to improve availability without risking poor performance due to unnecessary view changes.

## 8.2 File System Benchmarks

Next, we present the results of a set of experiments to evaluate the performance of a real service—BFS. The experiments compared the performance of BFS with two other implementations of NFS: NO-REP, which is identical to BFS except that it is not replicated, and NFS-STD, which is the NFS V2 implementation in Linux with Ext2fs at the server. The first comparison allows us to evaluate the overhead of the BFT library accurately within an implementation of a real service. The second comparison shows that BFS is practical: its performance is similar to the performance of NFS-STD, which is used daily by many users. Since the implementation of NFS in Linux does not ensure stability of modified data and metadata before replying to the client (as required by the NFS protocol [Sandberg et al. 1985]), we also compare BFS with NFS-DEC,

which is the NFS implementation in Digital UNIX and provides the correct semantics.

The section starts with a description of the experimental setup. Then it evaluates the performance of BFS without view changes or proactive recovery and it ends with an analysis of the cost of proactive recovery.

**8.2.1 Experimental Setup.** The experiments to evaluate BFS used the setup described in Section 8.1.1. They ran two well-known file system benchmarks: the modified Andrew benchmark [Ousterhout 1990; Howard et al. 1988] and PostMark [Katcher 1997].

The modified Andrew benchmark emulates a software development workload. It has several 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: Phases 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. We ran Andrew100 in a file system file with 205 MB and Andrew500 in a file system file with 1 GB; both benchmarks fill more than 90% of these files. Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

PostMark [Katcher 1997] models the load on Internet service providers. It emulates the workload generated by a combination of electronic mail, netnews, and Web-based commerce transactions. The benchmark starts by creating a large pool of files with random sizes within a configurable range. Then it runs a large number of transactions on these files. Each transaction consists of a pair of subtransactions: the first one creates or deletes a file, and the other one reads a file or appends data to a file. The operation types for each subtransaction are selected randomly with uniform probability distribution. After completing all the transactions, the remaining files are deleted.

We configured PostMark with an initial pool of 10,000 files with sizes between 512 bytes and 16 Kbytes. The files were uniformly distributed over 130 directories. The benchmark ran 100,000 transactions.

For all benchmarks and NFS implementations, 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, 4,096-byte read and write buffers, allowing write-back client caching, and allowing attribute caching. Both NO-REP and BFS used two relay processes at the client.

Out of the 18 operations in the NFS V2 protocol only `getattr` is read-only because the time-last-accessed attribute of files and directories is set by operations that would otherwise be read-only, for example, `read` and `lookup`. We modified BFS and NO-REP not to maintain the time-last-accessed attribute in order to apply the read-only optimization to `read` and `lookup` operations.

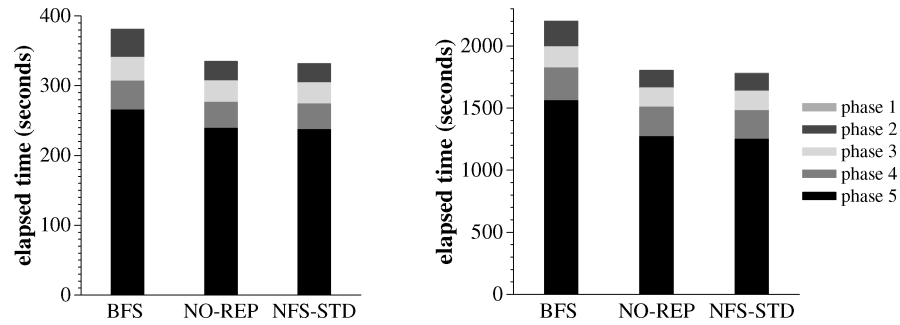


Fig. 15. Andrew100 and Andrew500: elapsed time in seconds.

This modification violates strict UNIX file system semantics but is unlikely to have adverse effects in practice.

**8.2.2 Performance Without Recovery.** We now analyze the performance of BFS without view changes or proactive recovery. We start by presenting results of experiments that ran with four replicas and later present results obtained with seven replicas.

*Andrew Benchmark.* Figure 15 presents results for Andrew100 and Andrew500 in a configuration with four replicas and one client machine. We report the mean of three 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 comparison between BFS and NO-REP shows that the overhead of Byzantine fault tolerance is low for this service—BFS takes only 14% more time to run Andrew100 and 22% more time to run Andrew500. This slowdown is smaller than the one measured with the microbenchmarks because the client spends a significant fraction of the elapsed time computing between operations, and operations at the server perform some computation. In addition, there are a significant number of disk writes at the server in Andrew500. The overhead is not uniform across the benchmark phases: it is 40% and 45% for the first two phases and approximately 11% for the last three. The main reason for this is a variation in the amount of time the client spends computing between operations.

The comparison with NFS-STD shows that BFS can be used in practice; it takes only 15% longer to complete Andrew100 and 24% longer to complete Andrew500. The performance difference would be smaller if Linux implemented NFS correctly. For example, the results in Castro [2001] show that BFS is 2% faster than the NFS implementation in Digital UNIX, which implements the correct semantics. The implementation of NFS on Linux does not ensure stability of modified data and metadata before replying to the client (as required by the NFS protocol), whereas BFS ensures stability through replication.

*PostMark.* Figure 16 presents the throughput measured using PostMark. The results are averages of three runs and the standard deviation was below 2%

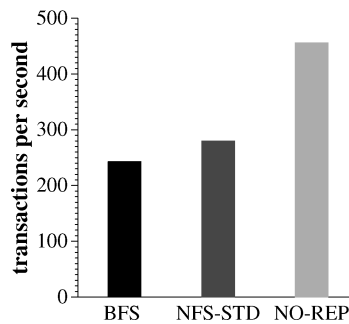


Fig. 16. PostMark: throughput in transactions per second.

of the reported value. The overhead of Byzantine fault tolerance is higher in this benchmark: BFS's throughput is 47% lower than NO-REP's. This is explained by a reduction on the computation time at the client relative to Andrew. What is interesting is that BFS's throughput is only 13% lower than NFS-STD's. The higher overhead is offset by an increase in the number of disk accesses performed by NFS-STD in this workload.

*More Replicas.* We also ran Andrew100 in a configuration with seven replicas ( $f = 2$ ). All replicas had a 600-MHz Pentium III processor and the client had a 700-MHz Pentium III processor. The results show that improving the resilience of the system by increasing the number of replicas from four to seven does not degrade performance significantly: BFS with  $f = 2$  is only 3% slower than with  $f = 1$ . This outcome was predictable given the microbenchmark results in the previous sections.

**8.2.3 Performance with 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 coprocessor, the read-only memory, and the watchdog timer in software. We are also simulating fast reboots. The LinuxBIOS project [Minnich 2000] 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`) [Minnich 2000]. 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).

*Recovery Time.* The time to complete recovery determines the minimum window of vulnerability that can be achieved without overlaps. We measured

Table II. Andrew: Maximum Recovery Time (seconds)

	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

the recovery time for Andrew100 and Andrew500 with 30-s reboots and with  $T_k = 15$  s between key changes.

Table II 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 II 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.

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 ran the experiments for Andrew100 with a 1-s reboot and the maximum time to complete recovery in this case was 13.3 s. 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. Although 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.

*Recovery Overhead.* We also evaluated the impact of recovery on performance in the experimental setup described in the previous section; Figure 17 shows the elapsed time to complete Andrew100 and Andrew500 as the window of vulnerability increases. BFS-PR is BFS with proactive recoveries. The number in square brackets is the minimum window of vulnerability in minutes.

The results show that adding frequent proactive recoveries to BFS has a low impact on performance: BFS-PR[4] is 16% slower than BFS in Andrew100

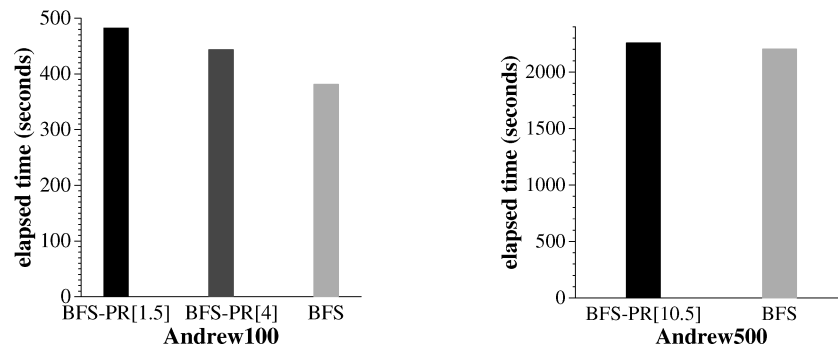


Fig. 17. Andrew: elapsed time in seconds with and without proactive recoveries.

and BFS-PR[1.5] is only 27% slower (even though every 15 s one replica starts a recovery). The overhead of proactive recovery in Andrew500 is even lower: BFS-PR[10.5] is 2% slower than BFS.

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.

## 9. RELATED WORK

There is a large body of research on replication but the earlier work did not provide an adequate solution for building systems that can tolerate software bugs, operator mistakes, or malicious attacks.

### 9.1 Replication with Benign Faults

Much research on replication has focused on techniques that tolerate benign faults (e.g., Alsberg and Day [1976], Gifford [1979], Schneider [1982], Oki and Liskov [1988], Lamport [1989], Liskov et al. [1991], and Keidar and Dolev [1996, 1998]): this work assumes replicas fail by stopping or by omitting some steps. This assumption is not valid with software bugs, operator mistakes, or malicious attacks. For example, an attacker can replace the code of a faulty replica to make it behave arbitrarily. Furthermore, services with mutable state may return incorrect replies when a single replica fails because this replica may propagate corrupt information to the others. Consequently, replication may decrease resilience to these types of faults because the probability of incorrect service behavior increases with the number of replicas.

Viewstamped replication [Oki and Liskov 1988] and Paxos [Lamport 1989] use a combination of primary-backup [Alsberg and Day 1976] and quorum [Gifford 1979] techniques to tolerate benign faults in an asynchronous system. They use a primary to assign sequence numbers to requests and they replace

primaries that appear to be faulty using a view-change protocol. Both algorithms use quorums to ensure that request ordering information is propagated to the new view. BFT borrows these ideas from the two algorithms but tolerating Byzantine faults requires a protocol that is significantly more complex.

## 9.2 Replication with Byzantine Faults

Techniques that tolerate Byzantine faults [Pease et al. 1980; Lamport et al. 1982] make no assumptions about the behavior of faulty components and, therefore, can tolerate even malicious attacks. However, most earlier work (e.g., Pease et al. [1980], Lamport et al. [1982], Schneider [1990], Cristian et al. [1985], Reiter [1996], Garay and Moses [1998], and Khilstrom et al. [1998]) assumes synchrony. This assumption is reasonable in some systems, for example, avionics control [Wensley et al. 1978]. But it is particularly dangerous when malicious attackers can launch denial-of-service attacks to flood the processors or the network with spurious requests.

**9.2.1 Agreement and Consensus.** Some agreement and consensus algorithms tolerate Byzantine faults in asynchronous systems (e.g., Bracha and Toueg [1985], Canetti and Rabin [1992], Malkhi and Reiter [1996b], Doudou et al. [1999], and Cachin et al. [2000]). However, they do not provide a complete solution for state machine replication and, furthermore, most of them are too slow to be used in practice.

BFT's protocol during normal case operation is similar to the Byzantine agreement algorithm in Bracha and Toueg [1985]. However, this algorithm is insufficient to implement state machine replication: it guarantees that non-faulty processes agree on a message sent by a primary but it is unable to survive primary failures.

**9.2.2 State Machine Replication.** Our work is inspired by Rampart [Reiter 1994, 1995, 1996; Malkhi and Reiter 1996a] and SecureRing [Khilstrom et al. 1998], which also implement state machine replication. However, these systems rely on synchrony assumptions for safety.

Both Rampart and SecureRing use group communication techniques with dynamic group membership. They must exclude faulty replicas from the group to make progress (e.g., to remove a faulty primary and elect a new one), and to perform garbage collection. For example, a replica is required to know that a message was received by all the replicas in the group before it can discard the message, so it may be necessary to exclude faulty nodes to discard messages.

These systems rely on failure detectors to determine which replicas are faulty. However, failure detectors cannot be accurate in an asynchronous system [Lynch 1996]; that is, they may misclassify a replica as faulty. Since correctness requires that fewer than  $1/3$  of group members be faulty, a misclassification can compromise correctness by removing a nonfaulty replica from the group. This opens an avenue of attack: an attacker gains control over a single replica but does not change its behavior in any detectable way; then it slows correct replicas or the communication between them until enough are excluded from the group. It is even possible for these systems to behave incorrectly



without any compromised replicas. This can happen if all the replicas that send a reply to a client are removed from the group and the remaining replicas never process the client's request.

To reduce the probability of misclassification, failure detectors can be calibrated to delay classifying a replica as faulty. However, for the probability to be negligible the delay must be very large, which is undesirable. For example, if the primary has actually failed, the group will be unable to process client requests until the delay has expired, which reduces availability. Our algorithm is not vulnerable to this problem because it only requires communication between quorums of replicas. Since there is always a quorum available with no faulty replicas, BFT never needs to exclude replicas from the group.

Public key cryptography was the major performance bottleneck in Rampart and SecureRing despite the fact that these systems include sophisticated techniques to reduce the cost of public key cryptography at the expense of security or latency. These systems rely on public key signatures to work correctly and cannot use symmetric cryptography to authenticate messages. BFT uses MACs to authenticate all messages and public key cryptography is used only to exchange the symmetric keys to compute the MACs. This approach improves performance by up to two orders of magnitude without losing security.

Rampart and SecureRing 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: the system may be unable to provide safety if a replica that is not faulty is removed from the group to be recovered; and 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 algorithm that we described in Castro and Liskov [1999b] and the algorithm in Doudou et al. [2000] are similar to BFT. They also work correctly in asynchronous systems but they rely on public key cryptography to sign messages. Therefore they perform poorly and do not support recovery. In addition, the algorithm in Doudou et al. [2000] does not provide garbage collection and state transfer mechanisms.

**9.2.3 Quorum Replication.** Phalanx [Malkhi and Reiter 1998a,b] and its successor Fleet [Malkhi and Reiter 2000] apply quorum replication techniques [Gifford 1979] 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 or write individual variables, and it offers consensus objects that can be used by clients to implement more complex operations. This makes Fleet more vulnerable to malicious clients because it relies on clients to group and order reads and writes to preserve any invariants over the service state. It is nontrivial for correct Fleet replicas to check invariants because they do not necessarily agree on the value of the state when they execute a write operation.

Fleet provides an algorithm with optimal resilience ( $n > 3f$  replicas to tolerate  $f$  faults) but malicious clients can make the state of correct replicas

diverge when this algorithm is used. To prevent this, Fleet requires  $n > 4f$  replicas.

Fleet does not provide a recovery mechanism for faulty replicas. However, it includes a mechanism to estimate the number of faulty replicas in the system [Alvisi et al. 1999] and a mechanism to adapt the threshold  $f$  on the number of faults tolerated by the system based on this estimate [Alvisi et al. 2000]. This is interesting but it is not clear whether it will work in practice: a clever attacker can make compromised replicas appear to behave correctly until it controls more than  $f$  and then it is too late to adapt or respond in any way.

There are no published performance numbers for Fleet or Phalanx but we believe our system is faster because it has fewer message delays in the critical path and because of our use of MACs rather than public key cryptography. In Fleet, writes require three message round trips to execute whereas BFT executes read-write operations in two round trips. More precisely, a write in Fleet requires three 1-to-many message exchanges and three many-to-1 message exchanges whereas in BFT read-write operations require two 1-to-many exchanges, one many-to-many exchange, and one many-to-1 exchange. Most reads in Fleet and read-only operations in BFT require one round trip and involve the same type of message exchanges.

In addition, all communication in Fleet is between the client and the replicas. This reduces opportunities for request batching and may result in increased latency since we expect that in most configurations communication between replicas will be faster than communication with the client.

The approach in Fleet offers the potential for improved scalability: each operation is processed by only a subset of replicas. However, the load on each replica decreases slowly with  $n$  (it is  $\Omega(1/\sqrt{n})$ ). Therefore we believe that client caching and partitioning the state by several replica groups is a better approach to achieve scalability for most applications.

There has been some recent work on augmenting Fleet with support for state machine replication [Chockler et al. 2001]. This work uses an algorithm similar to BFT with clients playing the role of primary. The algorithm assumes that clients are correct and it assumes eventual time bounds on delays for liveness but it is safe in asynchronous systems. It requires  $n > 5f$  replicas with public key signatures or  $n > 6f$  without signatures, and four round trips per operation.

COCA [Zhou et al. 2000] uses quorum replication techniques combined with proactive recovery to implement an online certification authority. Like BFT, it provides strong safety and liveness guarantees if fewer than  $1/3$  of the replicas fail within any window of vulnerability. COCA specifies the semantics of certificate operations carefully to be able to provide liveness without relying on any synchrony assumption. BFT must rely on a weak synchrony assumption for liveness due to its generality.

COCA's proactive recovery uses an interesting asynchronous proactive signature sharing mechanism to ensure that the certification authority's signing key is not compromised when replicas fail and recover. It does not rely on secure coprocessors to perform recoveries but it may need to involve administrators in the recovery of compromised replicas.

COCA provides defenses against denial-of-service attacks that are similar to those in BFT [Castro 2001]. COCA has been implemented and its performance has been evaluated with and without denial-of-service attacks. The performance is worse than BFT's due to extensive use of public key cryptography but some of this cryptography cannot be avoided with the certification authority specification used in COCA.

### 9.3 Other Related Work

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.

The SFS read-only file system [Fu et al. 2000] uses a technique to transfer data between replicas and clients that is similar to our state transfer technique. They are both based on Merkle trees [Merkle 1987] but the read-only SFS uses data structures that are optimized for a file system service. Another difference is that our state transfer handles modifications to the state while the transfer is in progress whereas their file system is read-only. Our technique to check the integrity of the replica's state during recovery is similar to those in Blum et al. [1994] and Maheshwari et al. [2000] except that we obtain the tree with correct digests from the other replicas rather than from a secure coprocessor.

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 Ostrovsky and Yung [1991]. This concept has previously been applied in synchronous systems to secret-sharing schemes [Herzberg et al. 1995], threshold cryptography [Herzberg et al. 1997], and more recently secure information storage and retrieval [Garay et al. 2000] (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.

## 10. CONCLUSION

The growing reliance of our society on computers demands highly available systems that provide correct service without interruptions. Byzantine faults such as software bugs, operator mistakes, and malicious attacks are the major cause of service interruptions. We present a new replication algorithm and implementation techniques to build highly available systems that tolerate Byzantine faults and can be used in practice.

This article describes BFT, a state machine replication algorithm that tolerates Byzantine faults provided fewer than  $1/3$  of the replicas are faulty. BFT provides linearizability, which is a strong safety property, without relying on any synchrony assumption. Additionally, it guarantees liveness provided message delays are bounded eventually. BFT provides safety and liveness regardless of the number of Byzantine-faulty clients.

The article also describes a proactive recovery mechanism that allows the replicated system 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. Replicas can be recovered frequently to shrink the window of vulnerability to a few minutes with a low impact on performance. The mechanism also provides detection of denial-of-service attacks aimed at increasing the window and detects when the state of a replica is corrupted by an attacker.

BFT has been implemented as a generic program library with a simple interface and the article describes a service that was implemented using the library: the first Byzantine-fault-tolerant NFS file system, BFS. The BFT library and BFS perform well. For example, BFS with four replicas performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated. This good performance is due to several optimizations. The most important optimization is the use of symmetric cryptography to authenticate messages. Public key cryptography, which was the major bottleneck in previous systems, is used only to exchange the symmetric keys.

## APPENDIX

This appendix presents a detailed formal specification of the BFT algorithm described in Section 4. We specified a simplified version of BFT to improve clarity. In particular, the formal specification omits code to defend against denial-of-service attacks aimed at consuming replicas' memory space, and code to ensure fair scheduling of requests. Our actual implementation ensures a constant bound on the amount of memory used and fair scheduling even in the presence of denial-of-service attacks. In addition, the specification uses simple but inefficient state transfer and retransmission strategies. Finally, it does not model the mechanism to trigger view changes and improve liveness; instead, each replica decides nondeterministically when to change to the next view.

The appendix starts by providing an overview of the system, and by defining BFT's safety property formally. Then it describes the models for the algorithms run by clients and replicas.

### A. OVERVIEW

We model the service replicated by BFT as a deterministic state machine, which is a tuple  $\langle S, \mathcal{U}, \mathcal{O}, \mathcal{O}', g, s_0 \rangle$ . It has a state in a set  $S$  (initially equal to  $s_0$ ) and its behavior is defined by a transition function:

$$g : \mathcal{U} \times \mathcal{O} \times S \rightarrow \mathcal{O}' \times S.$$

The arguments to the function are a client identifier in a set of users  $\mathcal{U}$ , an operation in a set  $\mathcal{O}$ , which encodes an operation identifier and any arguments to that operation, and an initial state. These arguments are mapped by  $g$  to the result of the operation in  $\mathcal{O}'$  and a new state. The client identifier is included explicitly as an argument to  $g$  because the algorithm authenticates the client that requests an operation and provides the service with its identity. This enables the service to enforce access control.

The distributed system that implements the replicated service is modeled as a set of I/O automata [Lynch 1996]. An I/O automaton has a state and a set of actions that define state transitions. Each action has a precondition, which determines whether it is enabled, and effects, which determine how the state is modified when it executes. The actions of an I/O automaton are classified as input, output, and internal actions, where input actions are required to be always enabled. Automata execute by repeating the following two steps: an enabled action is selected nondeterministically, and then it is executed atomically. Several automata can be composed by combining input and output actions. Lynch's book [Lynch 1996] provides a good description of I/O automata.

There is a proxy automaton  $P_c$  for each client  $c$ .  $P_c$  provides an input action for  $c$  to invoke an operation  $o$  on the state machine,  $\text{REQUEST}(o)_c$ , and an output action for  $c$  to learn the result  $r$  of an operation it requested,  $\text{REPLY}(r)_c$ .  $P_c$  communicates with a set of replicas to implement the interface it offers to the client. Each replica has a unique identifier  $i$  in a set  $\mathcal{R}$  and is modeled by an automaton  $R_i$ .

Replicas and proxies execute in different nodes in the distributed system. The network between replicas and proxies is an automaton with a  $\text{SEND}(m, N)_i$  and a  $\text{RECEIVE}(m)_i$  action for each proxy and replica  $i$ . These actions allow automata to send messages in a universal message set  $\mathcal{M}$  to any subset of automata with identifiers in  $\mathcal{N} = \mathcal{U} \cup \mathcal{R}$ . The assumptions about this network were discussed in Section 2.

We use the notation from Section 2 to denote message authentication. For example,  $m_{\alpha_i}$  denotes a message with a valid authenticator produced by  $i$ . Since a replica cannot verify the correctness of all the entries in authenticators it receives, we use the notation  $m_{\alpha_{ij}}$  to denote a message with an authenticator from  $i$  with a valid entry for  $j$ .

## B. SAFETY PROPERTY

The safety property offered by BFT is a form of linearizability [Herlihy and Wing 1987]: the replicated service behaves as a centralized implementation that executes operations atomically one at a time.

We modified the definition of linearizability because the original definition does not work with Byzantine-faulty clients. The problem is that these clients are not restricted to use the  $\text{REQUEST}$  and  $\text{REPLY}$  interface provided by the proxy automata. For example, they can make the replicated service execute their requests by injecting appropriate messages directly into the network. Therefore, the modified linearizability property treats faulty and nonfaulty clients differently.

A similar modification to linearizability was proposed concurrently in Malkhi et al. [1998]. Their proposal uses conditions on execution traces to specify the modified linearizability property. We specify the property using an I/O automaton, *Safe*, with the same external signature as the composition of the proxy automata. Our approach has several advantages: it produces a simpler specification and it enables the use of state-based proof techniques such as invariant assertions and simulation relations to reason about linearizability. These proof

<b>Signature:</b>	
Input:	REQUEST( $o$ ) <sub><math>c</math></sub> CLIENT-FAILURE <sub><math>c</math></sub> REPLICA-FAILURE <sub><math>i</math></sub>
Internal:	EXECUTE( $o, t, c$ ) FAULTY-REQUEST( $o, t, c$ )
Output:	REPLY( $r$ ) <sub><math>c</math></sub>
<b>State:</b>	
$val \in \mathcal{S}$ , initially $s_o$	
$in \subseteq \mathcal{O} \times \mathbb{N} \times \mathcal{U}$ , initially $\{\}$	
$out \subseteq \mathcal{O} \times \mathbb{N} \times \mathcal{U}$ , initially $\{\}$	
$\forall c \in \mathcal{U}$ , $last-req_c \in \mathbb{N}$ , initially $last-req_c = 0$	
$\forall c \in \mathcal{U}$ , $last-rep-t_c \in \mathbb{N}$ , initially $last-rep-t_c = 0$	
$\forall c \in \mathcal{U}$ , $faulty-client_c \in \text{Boolean}$ , initially $faulty-client_c = \text{false}$	
$\forall i \in \mathcal{R}$ , $faulty-replica_i \in \text{Boolean}$ , initially $faulty-replica_i = \text{false}$	
$n\text{-faulty} \equiv  \{i \mid faulty-replica_i = \text{true}\} $	
<b>Transitions</b> (if $n\text{-faulty} \leq \lfloor \frac{ \mathcal{R} -1}{3} \rfloor$ ):	
REQUEST( $o$ ) <sub><math>c</math></sub>	FAULTY-REQUEST( $o, t, c$ )
Eff: $last-req_c := last-req_c + 1$ $in := in \cup \{\langle o, last-req_c, c \rangle\}$	Pre: $faulty-client_c = \text{true}$ Eff: $in := in \cup \{\langle o, t, c \rangle\}$
CLIENT-FAILURE <sub><math>c</math></sub>	EXECUTE( $o, t, c$ )
Eff: $faulty-client_c := \text{true}$	Pre: $\langle o, t, c \rangle \in in$ Eff: $in := in - \{\langle o, t, c \rangle\}$
REPLICA-FAILURE <sub><math>i</math></sub>	if $t > last-rep-t_c$ then
Eff: $faulty-replica_i := \text{true}$	$(r, val) := g(c, o, val)$
REPLY( $r$ ) <sub><math>c</math></sub>	$out := out \cup \{\langle r, t, c \rangle\}$ $last-rep-t_c := t$
Pre: $faulty-client_c = \text{true} \vee \exists t : (\langle r, t, c \rangle \in out$	
Eff: $out := out - \{\langle r, t, c \rangle\}$	

Fig. 18. Specification of safe behavior, *Safe*. Here  $o \in \mathcal{O}$ ,  $t \in \mathbb{N}$ ,  $c \in \mathcal{U}$ ,  $i \in \mathcal{R}$ , and  $r \in \mathcal{O}'$ .

techniques are better than those that reason directly about execution traces because they are more stylized and better suited to produce automatic proofs.

The specification of modified linearizability, *Safe*, is a simple, abstract, centralized implementation of the state machine  $\langle \mathcal{S}, \mathcal{U}, \mathcal{O}, \mathcal{O}', g, s_o \rangle$  that is defined in Figure 18. We say that the replicated service (obtained by composing proxy, replica, and network automata) satisfies the safety property if it implements *Safe* according to the definition in Lynch [1996].

The state of *Safe* includes the following components:  $val$  is the current value of the state machine,  $in$  records requests to execute operations, and  $out$  records replies with operation results. Each  $last-req_c$  component is used to timestamp requests by client  $c$  to totally order them, and  $last-rep-t_c$  remembers the value of  $last-req_c$  that was associated with the last operation executed for  $c$ . The  $faulty-client_c$  and  $faulty-replica_i$  indicate which clients and replicas are faulty.

The CLIENT-FAILURE and REPLICA-FAILURE actions are used to model failures; they set the  $faulty-client_c$  or the  $faulty-replica_i$  variables to true. The REQUEST( $o$ ) <sub>$c$</sub>  actions increment  $last-req_c$  to obtain a new timestamp for the request, and add a triple to  $in$  with the requested operation  $o$ , the timestamp value  $last-req_c$ , and the client identifier. The FAULTY-REQUEST actions are similar. They model

execution of requests by faulty clients that bypass the external signature, for example, by injecting the appropriate messages into the multicast channel.

The  $\text{EXECUTE}(o, t, c)$  actions pick a request with a triple  $\langle o, t, c \rangle$  in  $in$  for execution and remove the triple from  $in$ . They execute the request only if the timestamp  $t$  is greater than the timestamp of the last request executed on  $c$ 's behalf. This models a well-formedness condition on nonfaulty clients: they are expected to wait for the reply to the last requested operation before they issue the next request. Otherwise, one of the requests may not even execute and the client may be unable to match the replies with the requests. When a request is executed, the transition function of the state machine  $g$  is used to compute a new value for the state and a result  $r$  for operation  $o$ . The client identifier is passed as an argument to  $g$  to allow the service to enforce access control. Then the actions add a triple with the result  $r$ , the request timestamp, and the client identifier to  $out$ .

The  $\text{REPLY}(r)_c$  actions return an operation result with a triple in  $out$  to client  $c$  and remove the triple from  $out$ . The  $\text{REPLY}$  precondition is weaker for faulty clients to allow arbitrary replies for such clients. The algorithm cannot guarantee safety if more than  $\lfloor (|\mathcal{R}| - 1)/3 \rfloor$  replicas are faulty. Therefore, the behavior of *Safe* is left unspecified in this case.

### C. PROXY AUTOMATON

The proxy automaton  $P_c$  is defined in Figure 19. The proxy remembers the last request sent to the replicas in  $out_c$  and it collects replies that match this request in  $in_c$ . It uses  $last-req_c$  to generate timestamps for requests. The  $\text{REQUEST}$  actions add a request for the argument operation to  $out_c$ . This request is sent on the network by the send actions and it is retransmitted until a reply is generated. The  $\text{RECEIVE}$  actions collect replies in  $in_c$  that match the request in  $out_c$ . Once there are more than  $f$  replies with the same  $r$  in  $in_c$ , the  $\text{REPLY}$  action becomes enabled and returns the result of the requested operation to the client.

### D. REPLICA AUTOMATON

Figure 20 defines the signature and state of replica automaton  $R_i$ . The state variables include the current value of the  $i$ 's copy of the state machine  $val_i$ , the last reply  $last-rep_i$  sent to each client, and the timestamps in those replies  $last-rep-t_i$ . There is also a set of checkpoints  $chkpts_i$ , whose elements contain not only a snapshot of  $val_i$  but also of  $last-rep_i$  and  $last-rep-t_i$ . The log with messages received or sent by  $i$  is stored in  $in_i$ , and  $out_i$  buffers messages that are about to be sent.  $\mathcal{P}_i$  and  $\mathcal{Q}_i$  are used during view changes as explained in Section 4.5. Replicas also maintain the current view number  $view_i$ , a flag that indicates whether the view change into  $view_i$  is complete  $active_i$ , the sequence number of the last request executed  $last-exec_i$ , and the last sequence number they picked for a request  $seqno_i$ .

Figure 20 also defines a few auxiliary functions. The most interesting are:  $in-w(n, i)$  that checks if  $n$  is between the low and high water marks in  $i$ 's log; and  $pre-prepared$ ,  $prepared$ , and  $committed$  that define the various states that client requests go through during the protocol (as explained in Section 4).

**Signature:**

Input:     REQUEST( $o$ )<sub>c</sub>  
           RECEIVE( $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$ )<sub>c</sub>

Output:    REPLY( $r$ )<sub>c</sub>  
           SEND( $m, N$ )<sub>c</sub>

**State:**

$in_c \subseteq \mathcal{M}$ , initially  $\{\}$   
 $out_c \subseteq \mathcal{M}$ , initially  $\{\}$   
 $last-req_c \in \mathbb{N}$ , initially 0

**Transitions:**

REQUEST( $o$ )<sub>c</sub>  
 Eff:  $last-req_c := last-req_c + 1$   
       $out_c := \{\langle \text{REQUEST}, o, last-req_c, c \rangle_{\alpha_c}\}$   
       $in_c := \{\}$

RECEIVE( $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$ )<sub>c</sub>  
 Eff: if  $(out_c \neq \{\}) \wedge last-req_c = t$  then  
       $in_c := in_c \cup \{\langle \text{REPLY}, v, t, c, i, r \rangle\}$

SEND( $m, \mathcal{R}$ )<sub>c</sub>  
 Pre:  $m \in out_c$   
 Eff: none

REPLY( $r$ )<sub>c</sub>  
 Pre:  $out_c \neq \{\} \wedge \exists R : (|R| > f \wedge \forall i \in R : (\exists v : (\langle \text{REPLY}, v, last-req_c, c, i, r \rangle \in in_c)))$   
 Eff:  $out_c := \{\}$

Fig. 19. Proxy automaton  $P_c$ : signature, state, and transitions. Here  $o \in \mathcal{O}$ ,  $v, t \in \mathbb{N}$ ,  $c \in \mathcal{U}$ ,  $i \in \mathcal{R}$ ,  $r \in \mathcal{O}'$ ,  $m \in \mathcal{M}$ ,  $R \subseteq \mathcal{R}$ , and  $N \subseteq \mathcal{N}$ .

Figure 21 presents the actions associated with the normal case protocol. The actions match the description in Section 4.3 closely. The execute action is the most complex. To ensure exactly once semantics, a replica executes a request only if its timestamp is greater than the timestamp in the last reply sent to the client. When it executes a request, the replica uses the state machine's transition function  $g$  to compute a new value for the state and a reply to send to the client. Then, if  $n \bmod K = 0$ , the replica takes a checkpoint by adding a snapshot of  $val_i$ ,  $last-rep_i$ , and  $last-rep-t_i$  to the checkpoint set and puts a matching CHECKPOINT message in  $out_i$  to be multicast to the other replicas.

Figure 22 presents the garbage collection actions. The RECEIVE action collects CHECKPOINT messages in the log and the COLLECT-GARBAGE action discards old messages and checkpoints when the replica has a stable certificate logged.

Section 4.5 presented a number of correctness conditions on VIEW-CHANGE and NEW-VIEW messages. These conditions are formalized in Figure 23. In particular, *correct- $\mathcal{X}$*  corresponds to the decision procedure in Figure 4.

The last set of actions is presented in Figure 24. The formalization follows the description in Section 4.5 closely but the last four actions deserve further explanation. The RETRANSMIT action retransmits the checkpoint and requests chosen by a valid NEW-VIEW message to any replicas that might be missing them. The two RECEIVE actions that follow are used by replicas to receive these retransmitted



**Signature:**

Input: RECEIVE( $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_{ci}}$ )  
 RECEIVE( $\langle \text{PRE-PREPARE}, v, n, d \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{PREPARE}, v, n, d, j \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{COMMIT}, v, n, j \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{CHECKPOINT}, n, d, j \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{VIEW-CHANGE}, v, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, j \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{VIEW-CHANGE-ACK}, v, j, k, d \rangle_{\mu_{ji}}$ )  
 RECEIVE( $\langle \text{NEW-VIEW}, v, \mathcal{V}, \mathcal{X} \rangle_{\alpha_{ji}}$ )  
 RECEIVE( $\langle \text{STATE}, h, s \rangle$ )  
 RECEIVE( $\langle \text{REQUEST}, o, t, c \rangle$ )

Internal: SEND-PRE-PREPARE( $m, v, n$ )<sub>i</sub>  
 SEND-PREPARE( $m, v, n$ )<sub>i</sub>  
 SEND-COMMIT( $m, v, n$ )<sub>i</sub>  
 EXECUTE( $m, v, n$ )<sub>i</sub>  
 COLLECT-GARBAGE( $n$ )<sub>i</sub>  
 VIEW-CHANGE( $v$ )<sub>i</sub>  
 SEND-NEW-VIEW( $\mathcal{V}, \mathcal{X}$ )<sub>i</sub>  
 PROCESS-NEW-VIEW( $\mathcal{V}, \mathcal{X}$ )<sub>i</sub>  
 RETRANSMIT( $\mathcal{V}, \mathcal{X}$ )<sub>i</sub>

Output: SEND( $m, N$ )<sub>i</sub>

**State:**

$val_i \in \mathcal{S}$ , initially  $s_o$   
 $last\_rep_i : \mathcal{U} \rightarrow \mathcal{O}'$ , initially  $\forall c \in \mathcal{U} : last\_rep_i(c) = nil$   
 $last\_rep\_t_i : \mathcal{U} \rightarrow \mathbb{N}$ , initially  $\forall c \in \mathcal{U} : last\_rep\_t_i(c) = 0$   
 $chkpts_i \subseteq \mathcal{S}' \equiv \mathcal{S} \times (\mathcal{U} \rightarrow \mathcal{O}') \times (\mathcal{U} \rightarrow \mathbb{N})$ , initially  $\{ \langle 0, \langle val_i, last\_rep_i, last\_rep\_t_i \rangle \} \}$   
 $in_i \subseteq \mathcal{M}$ , initially  $\{ null \}$   
 $out_i \subseteq \mathcal{M}$ , initially  $\{ \}$   
 $\mathcal{P}_i \subseteq \mathbb{N}^3$ , initially  $\{ \}$   
 $\mathcal{Q}_i \subseteq \mathbb{N}^3$ , initially  $\{ \}$   
 $view_i \in \mathbb{N}$ , initially 0  
 $active\_view_i \in \text{Boolean}$ , initially *true*  
 $last\_exec_i \in \mathbb{N}$ , initially 0  
 $seqno_i \in \mathbb{N}$ , initially 0  
 $h_i \equiv \min(\{ n \mid \langle n, s \rangle \in chkpts_i \})$

**Auxiliary functions:**

$tag(m, u) \equiv m = \langle u, \dots \rangle$   
 $primary(v) \equiv v \bmod |\mathcal{R}|$   
 $in\_w(n, i) \equiv 0 < n - h_i \leq L$   
 $in\_wv(v, n, i) \equiv in\_w(n, i) \wedge view_i = v$   
 $pre\_prepared(m, v, n, i) \equiv \langle n, D(m), v \rangle \in \mathcal{Q}_i \vee (m \in in_i \wedge \langle \text{PRE-PREPARE}, v, n, D(m) \rangle \in in_i)$   
 $prepared(m, v, n, i) \equiv \langle n, D(m), v \rangle \in \mathcal{P}_i$   
 $\vee (pre\_prepared(m, v, n, i) \wedge \exists R : (|R| \geq 2f \wedge \forall k \in R : (\langle \text{PREPARE}, v, n, D(m), k \rangle \in in_i)))$   
 $committed(m, v, n, i) \equiv prepared(m, v, n, i) \wedge$   
 $\exists R : (|R| \geq 2f + 1 \wedge \forall k \in R : (\langle \text{COMMIT}, v, n, k \rangle \in in_i))$

Fig. 20. Replica automaton  $R_i$ : signature, state, and auxiliary functions. Here  $t, v, n, h, d \in \mathbb{N}$ ,  $c \in \mathcal{U}$ ,  $i, j, k \in \mathcal{R}$ ,  $m \in \mathcal{M}$ ,  $s \in \mathcal{S}'$ ,  $\mathcal{V}, \mathcal{X}, \mathcal{C} \subseteq \mathbb{N}^2$ ,  $\mathcal{P}, \mathcal{Q} \subseteq \mathbb{N}^3$ , and  $N \subseteq \mathcal{N}$ .

```

RECEIVE( $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_{ci}}$ )i
  Eff: if  $t = \text{last-rep-}t_i(c)$  then
     $\text{out}_i := \text{out}_i \cup \{ \langle \text{REPLY}, \text{view}_i, t, c, i, \text{last-rep}_i(c) \rangle_{\mu_{ic}} \}$ 
  else if  $t > \text{last-rep-}t_i(c)$  then
     $\text{in}_i := \text{in}_i \cup \{ \langle \text{REQUEST}, o, t, c \rangle \}$ 

SEND-PRE-PREPARE( $m = \langle \text{REQUEST}, o, t, c \rangle, v, n$ )i
  Pre:  $\text{primary}(\text{view}_i) = i \wedge \text{seqno}_i = n - 1 \wedge \text{in-wv}(v, n, i) \wedge \text{active-view}_i \wedge$ 
     $m \in \text{in}_i \wedge \nexists \langle \text{PRE-PREPARE}, v, n', D(m) \rangle \in \text{in}_i$ 
  Eff:  $\text{seqno}_i := n$ 
    let  $p = \langle \text{PRE-PREPARE}, v, n, D(m) \rangle$ 
     $\text{out}_i := \text{out}_i \cup \{ p_{\alpha_i} \}$ 
     $\text{in}_i := \text{in}_i \cup \{ p \}$ 

RECEIVE( $\langle \text{PRE-PREPARE}, v, n, d \rangle_{\alpha_{ji}}$ )i ( $j \neq i$ )
  Eff: if  $j = \text{primary}(\text{view}_i) \wedge \text{in-wv}(v, n, i) \wedge \text{active-view}_i \wedge \nexists \langle \text{PRE-PREPARE}, v, n, d' \rangle \in \text{in}_i$  then
     $\text{in}_i := \text{in}_i \cup \{ \langle \text{PRE-PREPARE}, v, n, d \rangle \}$ 

SEND-PREPARE( $m, v, n$ )i
  Pre:  $\text{primary}(\text{view}_i) \neq i \wedge \text{pre-prepared}(m, v, n, i) \wedge \langle \text{PREPARE}, v, n, D(m), i \rangle \notin \text{in}_i$ 
  Eff: let  $p = \langle \text{PREPARE}, v, n, D(m), i \rangle$ 
     $\text{in}_i := \text{in}_i \cup \{ p \}$ 
     $\text{out}_i := \text{out}_i \cup \{ p_{\alpha_i} \}$ 

RECEIVE( $\langle \text{PREPARE}, v, n, d, j \rangle_{\alpha_{ji}}$ )i ( $j \neq i$ )
  Eff: if  $j \neq \text{primary}(\text{view}_i) \wedge \text{in-wv}(v, n, i)$  then
     $\text{in}_i := \text{in}_i \cup \{ \langle \text{PREPARE}, v, n, d, j \rangle \}$ 

SEND-COMMIT( $m, v, n$ )i
  Pre:  $\text{prepared}(m, v, n, i) \wedge \langle \text{COMMIT}, v, n, i \rangle \notin \text{in}_i$ 
  Eff: let  $c = \langle \text{COMMIT}, v, n, i \rangle$ 
     $\text{out}_i := \text{out}_i \cup \{ c_{\alpha_i} \}$ 
     $\text{in}_i := \text{in}_i \cup \{ c \}$ 

RECEIVE( $\langle \text{COMMIT}, v, n, j \rangle_{\alpha_{ji}}$ )i ( $j \neq i$ )
  Eff: if  $\text{in-wv}(v, n, i)$  then
     $\text{in}_i := \text{in}_i \cup \{ \langle \text{COMMIT}, v, n, j \rangle \}$ 

EXECUTE( $m, v, n$ )i
  Pre:  $n = \text{last-exec}_i + 1 \wedge \text{committed}(m, v, n, i)$ 
  Eff:  $\text{last-exec}_i := n$ 
    if ( $m \neq \text{null}$ ) then
      if  $\exists o, t, c : (m = \langle \text{REQUEST}, o, t, c \rangle \wedge t > \text{last-rep-}t_i(c))$  then
         $\text{last-rep-}t_i(c) := t$ 
         $(\text{last-rep}_i(c), \text{val}_i) := g(c, o, \text{val}_i)$ 
         $\text{out}_i := \text{out}_i \cup \{ \langle \text{REPLY}, \text{view}_i, t, c, i, \text{last-rep}_i(c) \rangle_{\mu_{ic}} \}$ 
      if  $n \bmod K = 0$  then
        let  $m' = \langle \text{CHECKPOINT}, n, D(\langle \text{val}_i, \text{last-rep}_i, \text{last-rep-}t_i \rangle), i \rangle$ 
         $\text{out}_i := \text{out}_i \cup \{ m'_{\alpha_i} \}$ 
         $\text{in}_i := \text{in}_i \cup \{ m' \}$ 
         $\text{chkpts}_i := \text{chkpts}_i \cup \{ \langle n, \langle \text{val}_i, \text{last-rep}_i, \text{last-rep-}t_i \rangle \rangle \}$ 

SEND( $m, N$ )i
  Pre:  $m \in \text{out}_i \wedge ((\neg \text{tag}(m, \text{REPLY}) \wedge N = \mathcal{R} - \{i\})$ 
     $\vee \exists v, t, c, r : (m = \langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}} \wedge N = \{c\}))$ 
  Eff:  $\text{out}_i := \text{out}_i - \{m\}$ 

```

 Fig. 21. Replica automaton  $R_i$ : normal case actions.

RECEIVE( $\langle \text{CHECKPOINT}, n, d, j \rangle_{\alpha_{ji}} \rangle_i$  ( $j \neq i$ )  
 Eff: if  $\text{in-w}(n, i)$  then  
 $\text{in}_i := \text{in}_i \cup \{ \langle \text{CHECKPOINT}, n, d, j \rangle \}$   
 COLLECT-GARBAGE( $n$ ) <sub>$i$</sub>   
 Pre:  $\exists R, d : (|R| > 2f \wedge i \in R \wedge \forall k \in R : (\langle \text{CHECKPOINT}, n, d, k \rangle \in \text{in}_i))$   
 Eff:  $\text{chkpts}_i := \text{chkpts}_i - \{ \langle n', s \rangle | n' < n \}$   
 $\text{in}_i := \text{in}_i - \{ \langle \text{PRE-PREPARE}, v, n', d \rangle | n' \leq n \}$   
 $\text{in}_i := \text{in}_i - \{ \langle \text{PREPARE}, v, n', d, j \rangle | n' \leq n \}$   
 $\text{in}_i := \text{in}_i - \{ \langle \text{COMMIT}, v, n', j \rangle | n' \leq n \}$   
 $\text{in}_i := \text{in}_i - \{ \langle \text{CHECKPOINT}, n', d, j \rangle | n' \leq n \}$   
 $\text{in}_i := \text{in}_i - \{ \langle \text{REQUEST}, o, t, c \rangle | \exists \langle n, \langle s, lr, lrt \rangle \rangle \in \text{chkpts}_i : (lrt(c) \geq t) \}$   
 $\mathcal{P}_i := \mathcal{P}_i - \{ \langle n', d, v \rangle | n' \leq n \}$   
 $\mathcal{Q}_i := \mathcal{Q}_i - \{ \langle n', d, v \rangle | n' \leq n \}$

Fig. 22. Replica automaton  $R_i$ : garbage collection actions.

$\text{correct-view-change}(m, v, j) \equiv$   
 $\exists \langle \text{VIEW-CHANGE}, v, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, j \rangle = m :$   
 $(\forall \langle n, d, v' \rangle \in \mathcal{P} \cup \mathcal{Q} : (v' < v \wedge n > h \wedge n \leq h + L) \wedge \forall \langle n, d \rangle \in \mathcal{C} : (n > h \wedge n \leq h + L))$   
 $\text{view}(m) \equiv \text{the first view number in } m$   
 $\text{correct-}\mathcal{V}(\mathcal{V}, i) \equiv$   
 $\forall \langle j, d \rangle \in \mathcal{V} :$   
 $\exists m \in \text{in}_i : (\text{tag}(m, \text{VIEW-CHANGE}) \wedge \text{view}(m) = \text{view}_i \wedge d = D(m)$   
 $\wedge (\text{primary}(\text{view}_i) \neq i \vee \exists R : (|R| > 2f - 2 \wedge \forall k \in R : (\langle \text{VIEW-CHANGE-ACK}, \text{view}_i, k, j, d \rangle \in \text{in}_i))))$   
 $\text{correct-}\mathcal{X}(\mathcal{X}, \mathcal{V}, i) \equiv$   
 let  $h = \min\{n | \exists d : (\langle n, d \rangle \in \mathcal{X})\}$  and  $V = \{m | \exists j : (\langle j, D(m) \rangle \in \mathcal{V}) \wedge m \in \text{in}_i\}$   
 $\forall n, d, d' : ((\langle n, d \rangle \in \mathcal{X} \wedge \langle n, d' \rangle \in \mathcal{X}) \Rightarrow d = d')$   
 $\wedge \forall \langle h, d \rangle \in \mathcal{X} :$   
 $\exists V_1, V_2 \subseteq V : (|V_1| > 2f \wedge |V_2| > f \wedge \forall m \in V_1 : (m.h \leq h) \wedge \forall m \in V_2 : (\langle h, d \rangle \in m.\mathcal{C}))$   
 $\wedge \forall h < n \leq h + L :$   
 $\exists \langle n, d \rangle \in \mathcal{X} :$   
 $(d \neq D(\text{null}) \wedge$   
 $\exists m \in V, V_1, V_2 \subseteq V, v \in \mathbb{N} : (\langle n, d, v \rangle \in m.\mathcal{P} \wedge |V_1| > 2f \wedge |V_2| > f$   
 $\wedge \forall m' \in V_1 : (m'.h < n \wedge \forall \langle n, d', v' \rangle \in m'.\mathcal{P} : (v' < v \vee (v' = v \wedge d' = d))))$   
 $\wedge \forall m' \in V_2 : \exists \langle n, d, v' \rangle \in m'.\mathcal{Q} : (v' \geq v))$   
 $\vee (d = D(\text{null}) \wedge$   
 $\exists V_1 \in V : (|V_1| > 2f \wedge \forall m' \in V_1 : (m'.h < n \wedge \forall \langle n', d', v' \rangle \in m'.\mathcal{P} : (n' \neq n))))$   
 $\text{correct-new-view}(\mathcal{X}, \mathcal{V}, i) \equiv$   
 $\text{correct-}\mathcal{V}(\mathcal{V}, i) \wedge \text{correct-}\mathcal{X}(\mathcal{X}, \mathcal{V}, i) \wedge \langle \text{NEW-VIEW}, \text{view}_i, \mathcal{V}, \mathcal{X} \rangle \in \text{in}_i$

Fig. 23. Replica automaton: auxiliary functions for view-change actions.

checkpoints or requests. These actions use the information in the VIEW-CHANGE and NEW-VIEW messages to check the correctness of the messages. Therefore, the messages do not need to be authenticated.

This retransmission strategy is simple but inefficient. In our actual implementation, replicas ask for requests that they are missing and they use the state transfer protocol from Section 6.2.2 to fetch missing checkpoints efficiently.

The PROCESS-NEW-VIEW action processes the NEW-VIEW message when the replica has a correct NEW-VIEW message, the checkpoint chosen in the message or a later one is stable at the replica, and the replica has all chosen requests with numbers greater than its stable checkpoint. This action makes the replica active in  $\text{view}_i$ ,

SEND-VIEW-CHANGE( $v$ )<sub>*i*</sub>  
 Pre:  $v = \text{view}_i + 1$   
 Eff:  $\text{view}_i := v$ ;  $\text{active-view}_i := \text{false}$   
 $\mathcal{P}_i := \{\langle n, D(m), v' \rangle \mid \text{prepared}(m, v', n, i) \wedge \forall m', v'' : (\neg \text{prepared}(m', v'', n, i) \vee v'' \leq v')\}$   
 $\mathcal{Q}_i := \{\langle n, D(m), v' \rangle \mid \text{pre-prepared}(m, v', n, i) \wedge \forall v'' : (\neg \text{pre-prepared}(m, v'', n, i) \vee v'' \leq v')\}$   
 let  $\mathcal{C} = \{\langle n, D(s) \rangle \mid \langle n, s \rangle \in \text{chkpts}_i\}$  and  $m = \langle \text{VIEW-CHANGE}, v, h_i, \mathcal{C}, \mathcal{P}_i, \mathcal{Q}_i, i \rangle$   
 $\text{out}_i := \text{out}_i \cup \{m_{\alpha_i}\}$   
 $\text{in}_i := \text{in}_i \cup \{m\} - \{m' \mid \text{view}(m') < v\}$

RECEIVE( $\langle \text{VIEW-CHANGE}, v, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, j \rangle_{\alpha_{ji}}$ ) ( $j \neq i$ )  
 Eff: let  $m = \langle \text{VIEW-CHANGE}, v, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, j \rangle$   
 if  $v \geq \text{view}_i \wedge \text{correct-view-change}(m, v, i) \wedge \neg \exists \langle \text{VIEW-CHANGE}, v, h', \mathcal{C}', \mathcal{P}', \mathcal{Q}', j \rangle \in \text{in}_i$  then  
 $\text{in}_i := \text{in}_i \cup \{m\}$   
 $\text{out}_i := \text{out}_i \cup \{\langle \text{VIEW-CHANGE-ACK}, v, i, j, D(m) \rangle_{\alpha_i}\}$

RECEIVE( $\langle \text{VIEW-CHANGE-ACK}, v, j, k, d \rangle_{\alpha_{ji}}$ ) ( $j \neq i$ )  
 Eff: if  $\neg \exists \langle \text{VIEW-CHANGE-ACK}, v, j, k, d' \rangle \in \text{in}_i$  then  
 $\text{in}_i := \text{in}_i \cup \langle \text{VIEW-CHANGE-ACK}, v, j, k, d \rangle$

SEND-NEW-VIEW( $\mathcal{V}, \mathcal{X}$ )<sub>*i*</sub>  
 Pre:  $\text{primary}(\text{view}_i) = i \wedge \text{correct-}\mathcal{V}(\mathcal{V}, i) \wedge \text{correct-}\mathcal{X}(\mathcal{X}, \mathcal{V}, i) \wedge \neg \exists \langle \text{NEW-VIEW}, \text{view}_i, \mathcal{V}', \mathcal{X}' \rangle \in \text{in}_i$   
 Eff:  $\text{in}_i := \text{in}_i \cup \{\langle \text{NEW-VIEW}, \text{view}_i, \mathcal{V}, \mathcal{X} \rangle\}$ ;  $\text{out}_i := \{\langle \text{NEW-VIEW}, \text{view}_i, \mathcal{V}, \mathcal{X} \rangle_{\alpha_i}\}$

RECEIVE( $\langle \text{NEW-VIEW}, v, \mathcal{V}, \mathcal{X} \rangle_{\alpha_{ji}}$ ) ( $j \neq i$ )  
 Eff: if  $v > 0 \wedge v \geq \text{view}_i \wedge j = \text{primary}(v) \wedge \neg \exists \langle \text{NEW-VIEW}, v, \mathcal{V}', \mathcal{X}' \rangle \in \text{in}_i$  then  
 $\text{in}_i := \text{in}_i \cup \{\langle \text{NEW-VIEW}, v, \mathcal{V}, \mathcal{X} \rangle\}$

RETRANSMIT( $\mathcal{V}, \mathcal{X}$ )<sub>*i*</sub>  
 Pre:  $\text{correct-new-view}(\mathcal{V}, \mathcal{X}, i)$   
 Eff: let  $h = \min\{n \mid \exists d : \langle n, d \rangle \in \mathcal{X}\}$   
 $\text{out}_i := \text{out}_i \cup \{\langle \text{STATE}, h, s \rangle \mid \langle h, s \rangle \in \text{chkpts}_i\}$   
 $\text{out}_i := \text{out}_i \cup \{m \mid \exists \langle n, d \rangle \in \mathcal{X} : (D(m) = d \wedge m \in \text{in}_i)\}$

RECEIVE( $\langle \text{REQUEST}, o, t, c \rangle$ )<sub>*i*</sub>  
 Eff: let  $m = \langle \text{REQUEST}, o, t, c \rangle$   
 if  $\exists \mathcal{V}, \mathcal{X} : (\text{correct-new-view}(\mathcal{V}, \mathcal{X}, i) \wedge \neg \text{active-view}_i \wedge \exists \langle n, D(m) \rangle \in \mathcal{X})$  then  
 $\text{in}_i := \text{in}_i \cup \{m\}$

RECEIVE( $\langle \text{STATE}, h, s \rangle$ )<sub>*i*</sub>  
 Eff: if  $\exists \mathcal{V}, \mathcal{X} : (\text{correct-new-view}(\mathcal{V}, \mathcal{X}, i) \wedge \neg \text{active-view}_i \wedge \langle h, D(s) \rangle \in \mathcal{X} \wedge h = \min\{n \mid \exists d : \langle n, d \rangle \in \mathcal{X}\})$  then  
 $\text{chkpts}_i := \text{chkpts}_i \cup \{\langle h, s \rangle\}$   
 if  $h > \text{last-exec}_i$  then  
 $(\text{val}_i, \text{last-rep}_i, \text{last-rep-}t_i) := s$ ;  $\text{last-exec}_i := h$   
 $\text{out}_i := \text{out}_i \cup \{\langle \text{CHECKPOINT}, h, D(s), i \rangle_{\alpha_i}\}$   
 $\text{in}_i := \text{in}_i \cup \{\langle \text{CHECKPOINT}, h, D(s), i \rangle\}$

PROCESS-NEW-VIEW( $\mathcal{V}, \mathcal{X}$ )<sub>*i*</sub>  
 Pre:  $\text{correct-new-view}(\mathcal{V}, \mathcal{X}) \wedge \neg \text{active-view}_i$   
 $\wedge h_i \geq \min\{n \mid \exists d : \langle n, d \rangle \in \mathcal{X}\} \wedge \forall \langle n, d \rangle \in \mathcal{X} : (n \leq h_i \vee \exists m \in \text{in}_i : (D(m) = d))$   
 Eff:  $\text{active-view}_i := \text{true}$   
 $\text{in}_i := \text{in}_i \cup \{\langle \text{PRE-PREPARE}, \text{view}_i, n, d \rangle \mid \langle n, d \rangle \in \mathcal{X} \wedge n > h_i\}$   
 $\text{seqno}_i := \max\{n \mid \langle \text{PRE-PREPARE}, \text{view}_i, n, d \rangle \in \text{in}_i\}$   
 if  $\text{primary}(\text{view}_i) \neq i$  then  
 $\text{in}_i := \text{in}_i \cup \{\langle \text{PREPARE}, \text{view}_i, n, d, i \rangle \mid \langle n, d \rangle \in \mathcal{X} \wedge n > h_i\}$   
 $\text{out}_i := \text{out}_i \cup \{\langle \text{PREPARE}, \text{view}_i, n, d, i \rangle \mid \langle n, d \rangle \in \mathcal{X}\}$

 Fig. 24. Replica automaton  $R_i$ : view-change actions.

and adds PRE-PREPARE messages for chosen requests to its log. Replicas other than the primary also send matching PREPARE messages.

#### ACKNOWLEDGMENTS

We would like to thank Fred Schneider, Marc Shapiro, and the anonymous referees for their helpful comments on drafts of this article.

#### REFERENCES

- ALSBERG, P. AND DAY, J. 1976. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, IEEE Computer Society Press, San Francisco, 627–644.
- ALVISI, L., MALKHI, D., PIERCE, E., REITER, M., AND WRIGHT, R. 2000. Dynamic Byzantine quorum systems. In *International Conference on Dependable Systems and Networks (DSN, FTCS-30 and DCCA-8)*, IEEE Computer Society Press, New York, 283–292.
- ALVISI, L., PIERCE, E., MALKHI, D., AND REITER, M. 1999. Fault detection for Byzantine quorum systems. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, IEEE Computer Society Press, San Jose, Calif. 357–371.
- BELLARE, M. AND MICCIANCIO, D. 1997. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology—EUROCRYPT’97, Lecture Notes in Computer Science*, vol. 1233, W. Fumy, Ed., Springer-Verlag, Konstanz, Germany, 163–192.
- BELLARE, M. AND ROGAWAY, P. 1995. Optimal asymmetric encryption—How to encrypt with RSA. In *Advances in Cryptology—EUROCRYPT’94, Lecture Notes in Computer Science*, vol. 950, A. D. Santis, Ed., Springer-Verlag, Perugia, Italy, 92–111.
- BELLARE, M. AND ROGAWAY, P. 1996. The exact security of digital signatures. How to sign with RSA and Rabin. In *Advances in Cryptology—EUROCRYPT’96, Lecture Notes in Computer Science*, vol. 1070, U. Maurer, Ed., Springer-Verlag, Zaragoza, Spain, 399–416.
- BENNETT, C., BESSETTE, F., BRASSARD, G., SALVAIL, L., AND SMOLIN, J. 1992. Experimental quantum cryptography. *J. Cryptol.* 5, 1, 3–28.
- BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., AND ROGAWAY, P. 1999. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO’99, Lecture Notes in Computer Science*, vol. 1666, M. Wiener, Ed., Springer-Verlag, Santa Barbara, Calif., 216–233.
- BLUM, M., EVANS, W., GEMMEL, P., KANNAN, S., AND NAOR, M. 1994. Checking the correctness of memories. *Algorithmica* 12, 225–244.
- BRACHA, G. AND TOUEG, S. 1985. Asynchronous consensus and broadcast protocols. *J. ACM* 32, 4, 824–240.
- CACHIN, C., KURSAWE, K., AND SHOUP, V. 2000. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*, ACM Press, Portland, Ore.
- CANETTI, R. AND RABIN, T. 1992. Optimal asynchronous byzantine agreement. Tech. Rep. #92-15, Computer Science Department, Hebrew University.
- CANETTI, R., HALEVI, S., AND HERZBERG, A. 1997. Maintaining authenticated communication in the presence of break-ins. In *Proceedings of the Fourth ACM Conference on Computers and Communication Security*, ACM Press, Zurich, Switzerland.
- CASTRO, M. 2001. Practical Byzantine fault tolerance. Tech. Rep. MIT/LCS/TR-817, MIT Laboratory for Computer Science. January.
- CASTRO, M. AND LISKOV, B. 1999a. A Correctness proof for a practical byzantine-fault-tolerant replication algorithm. Tech. Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science.
- CASTRO, M. AND LISKOV, B. 1999b. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX, New Orleans.
- CHOCKLER, G., MALKHI, D., AND REITER, M. 2001. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Phoenix, Ariz.

- CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. 1985. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Conference on Fault Tolerant Computing*, IEEE Computer Society Press, Ann Arbor, Mich.
- DEERING, S. AND CHERITON, D. 1990. Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Comput. Syst.* 8, 2 (May), 85–110.
- DOUDOU, A., GARBINATO, B., AND GUERRAOU, R. 2000. Modular abstractions for devising Byzantine-resilient state machine Replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, Nurnberg, Germany, 144–153.
- DOUDOU, A., GARBINATO, B., GUERRAOU, R., AND SCHIPER, A. 1999. Muteness failure detectors: Specification and implementation. In *Proceedings of the Third European Dependable Computing Conference (EDCC-3)*, *Lecture Notes in Computer Science*, vol. 1667, J. Hlavicka, E. Maehle, and A. Pataricza, Eds., Springer-Verlag, Prague, Czech Republic, 71–87.
- FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April), 374–382.
- FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. 2000. Fast and secure distributed read-only file system. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, USENIX, San Diego.
- GARAY, J. AND MOSES, Y. 1998. Fully polynomial Byzantine agreement for  $n > 3t$  processors in  $t + 1$  rounds. *SIAM J. Comput.* 27, 1 (Feb.), 247–290.
- GARAY, J., GENNARO, R., JUTLA, C., AND RABIN, T. 2000. Secure distributed storage and retrieval. *Theo. Comput. Sci.* 243, 1–2 (July), 363–389.
- GIFFORD, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM Press, Pacific Grove, Calif., 150–162.
- GONG, L. 1992. A security risk of depending on synchronized clocks. *Oper. Syst. Rev.* 26, 1 (Jan.), 49–53.
- GRAY, J. 2000. FT 101. Talk at the University of California at Berkeley.
- HERLIHY, M. P. AND WING, J. M. 1987. Axioms for concurrent objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, ACM Press, Munich, 13–26.
- HERZBERG, A., JAKOBSSON, M., JARECKI, S., KRAWCZYK, H., AND YUNG, M. 1997. Proactive public key and signature systems. In *Proceedings of the Fourth ACM Conference on Computers and Communication Security*, ACM Press, Zurich, Switzerland.
- HERZBERG, A., JARECKI, S., KRAWCZYK, H., AND YUNG, M. 1995. Proactive secret sharing, or: How to cope with perpetual leakage. In *Advances in Cryptology—CRYPTO'95, Lecture Notes in Computer Science*, vol. 963, D. Coppersmith, Ed., Springer-Verlag, Santa Barbara, Calif.
- HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.* 6, 1 (Feb.), 51–81.
- KATCHER, J. 1997. PostMark: A new file system benchmark. Tech. Rep. TR-3022, Network Appliance. October.
- KEIDAR, I. AND DOLEV, D. 1996. Efficient message ordering in dynamic networks. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, ACM Press, Philadelphia, 68–76.
- KEIDAR, I. AND DOLEV, D. 1998. Increasing the resilience of distributed and replicated database systems. *J. Computer Syst. Sci.* 57, 3 (Dec.), 309–324.
- KIHLSTROM, K., MOSER, L., AND MELLIAR-SMITH, P. 1998. The SecureRing protocols for securing group communication. In *Proceedings of the Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Hawaii.
- LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* 3, 2 (Nov.), 125–143.
- LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July), 558–565.
- LAMPORT, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. and Syst.* 6, 2 (Apr.), 254–280.
- LAMPORT, L. 1989. The part-time parliament. Research Rep. 49, Digital Equipment Corporation Systems Research Center, Palo Alto, Sept.

- LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July), 382–401.
- LAMPSON, B. 2001. The ABCDs of Paxos. Presented at *Principles of Distributed Computing*. Available at <http://www.research.microsoft.com/lampson>.
- LISKOV, B. AND ZILLES, S. 1975. Specification techniques for data abstractions. *IEEE Trans. Softw. Eng. SE-1*, 1 (Mar.), 7–17.
- LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. 1991. Replication in the Harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP)*, ACM Press, Pacific Grove, Calif., 226–238.
- LYNCH, N. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, Calif.
- MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, B. 2000. How to build a trusted database system on untrusted storage. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, USENIX, San Diego.
- MALKHI, D. AND REITER, M. 1996a. A high-throughput secure reliable multicast protocol. In *Proceedings of the Ninth Computer Security Foundations Workshop*, IEEE Computer Society Press, Ireland, 9–17.
- MALKHI, D. AND REITER, M. 1996b. Unreliable intrusion detection in distributed computations. In *Proceedings of the Ninth Computer Security Foundations Workshop*, IEEE Computer Society Press, Ireland, 9–17.
- MALKHI, D. AND REITER, M. 1998a. Byzantine quorum systems. *J. Distrib. Comput.* 11, 4, 203–213.
- MALKHI, D. AND REITER, M. 1998b. Secure and scalable replication in phalanx. In *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, West Lafayette, Ind.
- MALKHI, D. AND REITER, M. 2000. An architecture for survivable coordination in large distributed systems. *IEEE Trans. Knowl. Data Eng.* 12, 2 (Apr.), 187–202.
- MALKHI, D., REITER, M., AND LYNCH, N. 1998. A correctness condition for memory shared by Byzantine processes (Submitted).
- MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. 1999. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, ACM Press, Kiawah Island, S.C.
- MERKLE, R. 1987. A digital signature based on a conventional encryption function. In *Advances in Cryptology—Crypto’87, Lecture Notes in Computer Science*, vol. 293, C. Pomerance, Ed., Springer-Verlag, Santa Barbara, Calif., 369–378.
- MINNICH, R. 2000. The Linux BIOS home page. Available at <http://www.acl.lanl.gov/linuxbios>.
- MURPHY, B. AND LEVIDOW, B. 2000. Windows 2000 dependability. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, IEEE Computer Society Press, New York.
- OKI, B. AND LISKOV, B. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing*, ACM Press, Toronto, 8–17.
- OSTROVSKY, R. AND YUNG, M. 1991. How to withstand mobile virus attack. In *Proceedings of the Nineteenth Symposium on Principles of Distributed Computing*, ACM Press, Montreal, 51–59.
- OSTERHOUT, J. 1990. Why aren’t operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer Conference*, USENIX, Anaheim, Calif., 247–256.
- PEASE, M., SHOSTAK, R., AND LAMPORT, L. 1980. Reaching agreement in the presence of faults. *J. ACM* 27, 2 (April), 228–234.
- POSTEL, J. 1980. User datagram protocol. DARPA-Internet RFC-768.
- REITER, M. 1994. Secure agreement protocols. In *Proceedings of the Second ACM Conference on Computer and Communication Security*, ACM Press, Fairfax, Va., 68–80.
- REITER, M. 1995. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems. Lecture Notes in Computer Science*, vol. 938, Springer Verlag, New York, 99–110.
- REITER, M. 1996. A secure group membership protocol. *IEEE Trans. Softw. Eng.* 22, 1 (Jan.), 31–42.
- RIVEST, R. 1992. The MD5 message-digest algorithm. Internet RFC-1321.

- RODRIGUES, R., CASTRO, M., AND LISKOV, B. 2001. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the Eighteenth Symposium on Operating System Principles*, ACM Press, Banff, Canada.
- SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, USENIX, Portland, Ore, 119–130.
- SCHNEIDER, F. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec.), 299–319.
- SCHNEIDER, F. 1982. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.* 4, 2 (Apr.), 125–148.
- SCHNEIER, B. 1996. *Applied Cryptography*. Wiley, New York.
- SHA1 1994. Announcement of Weakness in Secure Hash Standard.
- WENSLEY, J., LAMPORT, L., GOLDBERG, J., GREEN, M., LEVITT, K., MELLAR-SMITH, M., SHOSTAK, R., AND WEINSTOCK, C. 1978. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE* 66, 10 (Oct.), 1240–1255.
- ZHOU, L., SCHNEIDER, F., AND RENESSE, R. 2000. COCA: A secure distributed on-line certification authority. Tech. Rep. 2000-1828, Department of Computer Science, Cornell University, Ithaca, NY., Dec. *ACM Trans. Comput. Syst.* (to appear).

Received February 2001; revised May 2002; accepted June 2002