Practical Byzantine Fault Tolerance

Miguel Castro

January 31, 2001

This research was supported in part by DARPA under contract DABT63-95-C-005, monitored by Army Fort Huachuca, and under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. The author was supported by a fellowship from the Portuguese Ministry for Science and Technology, and by a fellowship from the Calouste Gulbenkian Foundation.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts, USA

Practical Byzantine Fault Tolerance

by
Miguel Castro

Abstract

Our growing reliance on online services accessible on the Internet 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. This thesis describes a new replication algorithm, BFT, that can be used to build highly-available systems that tolerate Byzantine faults. It shows, for the first time, how to build Byzantine-fault-tolerant systems that can be used in practice to implement real services because they do not rely on unrealistic assumptions and they perform well. BFT works in asynchronous environments like 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. The window may increase under a denial-of-service attack but the algorithm can detect and respond to such attacks and it can also detect when the state of a replica is corrupted by an attacker.

BFT has been implemented as a generic program library with a simple interface. The BFT library provides a complete solution to the problem of building real services that tolerate Byzantine faults. 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 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. The performance results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated. Therefore, we believe that the BFT library can be used to build practical systems that tolerate Byzantine faults.

Keywords: algorithms, analytic modelling, asynchronous systems, Byzantine faults, correctness proofs, fault tolerance, high availability, integrity, performance, proactive security, replication, and security.

This report is a minor revision of the dissertation of the same title submitted to the Department of Electrical Engineering and Computer Science on November 30, 2000, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in that department. The thesis was supervised by Professor Barbara Liskov.

Acknowledgments

First, I must thank my thesis supervisor, Barbara Liskov, for her constant support and wise advice. I feel very fortunate for having had the chance to work closely with her.

The other members of my thesis committee, Frans Kaashoek, Butler Lampson, and Nancy Lynch suggested many important improvements to this thesis and interesting directions for future work. I greatly appreciate their suggestions.

It has been a pleasure to be a graduate student in the Programming Methodology Group. I want to thank all the group members: Atul Adya, Sarah Ahmed, Sameer Ajmani, Ron Bodkin, Philip Bogle, Chandrasekhar Boyapati, Dorothy Curtis, Sanjay Ghemawat, Robert Gruber, Kyle Jamieson, Paul Jonhson, Umesh Maheshwari, Andrew Myers, Tony Ng, Rodrigo Rodrigues, Liuba Shrira, Ziqiang Tang, Zheng Yang, Yan Zhang, and Quinton Zondervan. Andrew and Atul deserve special thanks for the many stimulating discussions we had. I also want to thank Rodrigo for reading my formal proof, and for his help in handling the details of the thesis submission process.

I am grateful to my parents for their support over the years. My mother was always willing to drop everything and cross the ocean to help us, and my father is largely responsible for my interest in computers and programming.

Above all, I want to thank my wife, Inês, and my children, Madalena, and Gonçalo. They made my life at MIT great. I felt so miserable without them during my last two months at MIT that I had to finish my thesis and leave.

Contents

Chapter 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 idea is simple: instead of using a single server to implement a service, these techniques replicate the server and use an algorithm to coordinate the replicas. The algorithm provides the abstraction of a single service to the clients but the replicated server continues to provide correct service even when a fraction of the replicas fail. Therefore, the system is highly available provided the replicas are not likely to fail all at the same time.

The problem is that research on replication has focused on techniques that tolerate *benign faults* (e.g., [AD76, Gif79, OL88, Lam89, LGG$^+$91]): these techniques assume components fail by stopping or by omitting some steps and may not provide correct service if a single faulty component violates this assumption. Unfortunately, this assumption is no longer valid because malicious attacks, operator mistakes, and software errors can cause faulty nodes to exhibit arbitrary behavior and they are increasingly common causes of failure. 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 [ML00]. In addition, the number of software errors is increasing due to the growth in size and complexity of software.

Techniques that tolerate *Byzantine faults* [PSL80, LSP82] provide a potential solution to this problem because they make no assumptions about the behavior of faulty components. There is a significant body of work on agreement and replication techniques that tolerate Byzantine faults. However, most earlier work (e.g., [CR92, Rei96, MR96a, MR96b, GM98, KMMS98]) either concerns techniques designed to demonstrate theoretical feasibility that are too inefficient to be used in practice, or relies on unrealistic assumptions that can be invalidated easily by an attacker. For example, it is dangerous to rely on *synchrony* for correctness, i.e., to rely on known bounds on message delays and process speeds. An attacker may compromise the correctness of a service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the replica group. Such a denial-of-service attack is generally easier than gaining control over a non-faulty node.

This thesis describes a new algorithm and implementation techniques to build highly-available systems that tolerate Byzantine faults. These systems can be used in practice because they perform well and do not rely on unrealistic assumptions. The next section describes our contributions in more detail.

1.1 Contributions

This thesis presents BFT, a new algorithm for state machine replication [Lam78, Sch90] that tolerates Byzantine faults. BFT offers both liveness and safety provided at most $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas are faulty. This means that clients eventually receive replies to their requests and those replies are correct according to linearizability [HW87, CL99a]. We used formal methods to specify the algorithm and prove its safety. Formal reasoning is an important step towards correctness because algorithms that tolerate Byzantine faults are subtle.

BFT is the first Byzantine-fault-tolerant, state-machine replication algorithm that works correctly in asynchronous systems like 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.

Safety is provided regardless of how many faulty clients are using the service (even if they collude with faulty replicas): all operations performed by faulty clients are observed in a consistent way by non-faulty clients. 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. The safety property ensures faulty clients are unable to break these invariants or bypass access controls. Algorithms that restrict service operations to simple reads and blind writes (e.g., [MR98b]) are more vulnerable to Byzantine-faulty clients; they rely on the clients to order and group these simple operations correctly in order to enforce invariants.

BFT is also the first Byzantine-fault-tolerant replication algorithm to recover replicas *proactively* in an asynchronous system; replicas are recovered 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. Limiting the number of failures that can occur in a finite window is a synchrony assumption but such an assumption is unavoidable: since Byzantine-faulty replicas can discard the service state, we must bound the number of failures that can occur before recovery completes. To tolerate less than $1/3$ faults over the lifetime of the system, we require no synchrony assumptions for safety.

The window of vulnerability can be made very small (e.g., a few minutes) under normal conditions with a low impact on performance. Our algorithm provides *detection* of denial-of-service attacks aimed at increasing the window; replicas can time how long a recovery takes and alert their administrator if it exceeds some pre-established bound. Therefore, integrity can be preserved even when there is a denial-of-service attack. Additionally, the algorithm detects when the state of a replica is corrupted by an attacker.

Unlike prior research in Byzantine fault tolerance in asynchronous systems, this thesis describes a complete solution to the problem of building real services that tolerate Byzantine faults. For example, it describes efficient techniques to garbage collect information, to transfer state to bring replicas up-to-date, to retransmit messages, and to handle services with non-deterministic behavior.

Additionally, 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 [Rei94] and throughput [MR96a] 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 return sizes increase.

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 thesis 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 thesis presents a thorough performance analysis of the BFT library and BFS. This analysis includes a detailed analytic performance model. 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 support our claim that the BFT library can be used to implement practical Byzantine-fault-tolerant systems.

There is one problem that deserves further attention: the BFT library (or any other replication technique) provides little benefit when there is a strong positive correlation between the failure probabilities of the different replicas. Our library is effective at masking several important types of faults, e.g., it can mask non-deterministic software errors and faults due to resource leaks.

Additionally, it can mask other types of faults if some simple steps are taken to increase diversity in the execution environment. For example, the library can mask administrator attacks or mistakes if replicas are administered by different people.

However, it is important to develop affordable and effective techniques to further reduce the probability of $1/3$ or more faults within the same window of vulnerability. In the future, we plan to explore existing independent implementations of important services like databases or file systems to mask additional types of faults. Chapter 10 discusses these issues in more detail.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 describes BFT-PK, which is a version of BFT that uses public-key signatures to authenticate all messages. We start by describing BFT-PK because it is simpler than BFT but captures the key ideas. This chapter presents a formalization of BFT-PK and Appendix A presents a formal safety proof. Chapter 3 describes BFT: it explains how to modify BFT-PK to use symmetric cryptography to authenticate all messages. The proactive recovery mechanism is presented in Chapter 4. Chapter 5 describes optimizations and implementation techniques that are important to implement a complete, practical solution for replication in the presence of Byzantine faults. The implementation of the BFT library and BFS is presented in Chapter 6. The analytic performance model is described in Chapter 7 and Chapter 8 presents a detailed performance analysis for the BFT library and BFS. Chapter 9 discusses related work. Finally, our conclusions and some directions for future work appear on Chapter 10.

Chapter 2

BFT-PK: An Algorithm With Signatures

This chapter describes BFT-PK, which is an algorithm that uses public-key signatures to authenticate all messages and does not support recovery. We start by explaining BFT-PK because it is simple and it captures the key ideas behind our more complex algorithms. The next chapters explain how to eliminate public-key signatures and perform recovery, and Chapter 5 describes several important optimizations.

We begin by describing our system model and assumptions. Section 2.2 describes the problem solved by the algorithm and states correctness conditions. The algorithm is described informally in Section 2.3 and Section 2.4 presents a formalization of the system model, the problem, and the algorithm. BFT-PK was first presented in [CL99c] and the formalization appeared in [CL99a].

2.1 System Model

Section 2.4.2 presents a formal definition of the system model. This section describes the model informally. BFT-PK is a form of *state machine* replication [Lam78, Sch90]: it can be used to replicate any service that can be modeled as a deterministic state machine. These services can have operations that perform arbitrary computations provided they are deterministic: the result and new state produced when an operation is executed must be completely determined by the current state and the operation arguments. We can handle some common forms of non-determinism as explained in Section 5.4. The idea is to modify the services to remove computations that make non-deterministic choices and to pass the results of those choices as operation arguments.

The algorithm does not require all replicas to run the same service code. It is sufficient for them to run implementations with the same observable behavior, that is, implementations that produce the same sequence of results for any sequence of operations they execute. A consequence of this observation is that service implementations are allowed to have non-deterministic behavior provided it is not observable. The ability to run different implementations or implementations with non-deterministic behavior is important to reduce the probability of simultaneous failures due to software errors.

The replicated service is implemented by n replicas. Clients issue requests to the replicated service to invoke operations and wait for replies. Clients and replicas are correct if they follow the algorithm in Section 2.3. The clients and replicas run in different nodes in an asynchronous distributed system. These nodes are connected by an unreliable network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

BFT-PK uses digital signatures. Any non-faulty client or replica, x, can authenticate messages it sends on the multicast channel by signing them. We denote a message m signed by x as $\langle m \rangle_{\sigma_x}$. The algorithm also uses a cryptographic hash function D to compute message digests.

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily. We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. But we 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 signature scheme is non-existentially forgeable even with an adaptive chosen message attack [GMR88]: if a node x is not faulty and it did not sign message m, the adversary is unable to generate a valid signature $\langle m \rangle_{\sigma_x}$ for any m. We also assume that the cryptographic hash function is collision resistant [Dam89]: 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 [BR96, Riv92]. Therefore, we will assume that they hold with probability one in the rest of the text.

If we were only concerned with non-malicious faults (e.g., software errors), it would be possible to relax the assumptions about the cryptographic primitives and use weaker, more efficient constructions.

2.2 Service Properties

BFT-PK provides both *safety* and *liveness* properties [Lyn96] assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty over the lifetime of the system. The safety property is a form of linearizability [HW87]: the replicated service behaves like a centralized implementation that executes operations atomically one at a time. We modified the original definition of linearizability because it does not work with Byzantine-faulty clients. Section 2.4.3 presents our modified definition formally.

In a fail-stop [SS83] model, it is possible to provide safety even when all replicas fail. But, in a Byzantine failure model, safety requires a bound on the number of faulty replicas because they can behave arbitrarily (for example, they can destroy their state).

The resilience of BFT-PK is optimal: $3f + 1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty. 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 replicated service may have to return a reply to a request before the request is received by all replicas. Since f replicas might be faulty and not responding, the service may have to return a reply before the request is received by more than $n - f$ replicas. 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 $n - 3f$ non-faulty replicas. It is impossible to ensure that the read returns the correct value unless R and W have at least one non-faulty 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 non-faulty clients. In particular, if the service operations are designed to preserve some invariants on the service state, faulty clients cannot break those invariants. This is an important defense against Byzantine-faulty clients that is enabled by BFT-PK's ability to implement an arbitrary *abstract data type* [LZ75] with complex operations.

Algorithms that restrict service operations to simple reads and blind writes (e.g., [MR98b]) are more vulnerable to Byzantine-faulty clients; they rely on the clients to order and group these simple operations correctly in order to enforce invariants. For example, creating a file requires updates to meta-data information. In BFT-PK, this operation can be implemented to enforce meta-data invariants such as ensuring the file is assigned a new inode. In algorithms that restrict the complexity of service operations, a faulty client will be able to write meta-data information and violate important invariants, e.g., it could assign the inode of another file to the newly created file.

The modified linearizability property may be insufficient to guard against faulty clients, e.g., in a file system a faulty client can write garbage data to some shared file. However, we 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. Also, services may provide operations to change the access permissions for a client. Since the algorithm ensures that the effects of access revocation operations are observed consistently by all clients, this provides a powerful mechanism to recover from attacks by faulty clients.

BFT-PK 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 [FLP85]. We guarantee liveness, i.e., clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty and *delay*(t) does not grow faster than t indefinitely. Here, *delay*(t) is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination (assuming the sender keeps retransmitting the message until it is received). This is a rather weak synchrony assumption that is likely to be true in any real system provided network faults are eventually repaired and denial-of-service attacks eventually stop, yet it enables us to circumvent the impossibility result in [FLP85].

There are randomized algorithms to solve consensus with Byzantine faults that do not rely on any synchrony assumption but provide probabilistic liveness guarantees, e.g.,[BT85, CR92, CKS00]. The algorithm in [BT85] assumes there is some round in which messages from correct replicas are delivered before the ones from faulty replicas; this is less likely to be true in practice than our synchrony assumption. The algorithms in [CR92, CKS00] do not rely on this assumption but, like BFT-PK, they are not going to be able to make progress in the presence of a network failure or denial-of-service attack that prevents communication among a majority of the replicas. Furthermore, they rely on expensive cryptography whereas we explain how to modify BFT-PK to use only inexpensive symmetric cryptography in Chapter 4.

Our algorithms do not address the problem of fault-tolerant privacy: a faulty replica may leak information to an attacker. It is not feasible to offer fault-tolerant privacy in the general case because service operations may perform arbitrary computations using their arguments and the service state; replicas need this information in the clear to execute such operations efficiently. It is possible to use secret sharing schemes [Sha79] to obtain privacy even in the presence of a threshold of malicious replicas [HT88] for the arguments and portions of the state that are opaque to the service operations. We plan to investigate these techniques in the future.

2.3 The Algorithm

Our algorithm builds on previous work on state machine replication [Lam78, Sch90]. 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. We denote the set of replicas by $\mathcal{R}$ and identify each replica using an integer in $\{0, ..., |\mathcal{R}| - 1\}$. For simplicity, we assume $|\mathcal{R}| = 3f + 1$ where f is the maximum number of replicas that may be faulty; although there could be more than $3f + 1$ replicas, the additional replicas degrade performance (since more and bigger messages are being exchanged) without providing improved resilience.

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

The hard problem in state machine replication is ensuring non-faulty replicas execute the same requests in the same order. Like Viewstamped Replication [OL88] and Paxos [Lam89], our algorithm uses a combination of primary-backup [AD76] and quorum replication [Gif79] 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*. We choose the primary of a view to be replica p such that $p = v \ mod \ |\mathcal{R}|$, where v is the view number and views are numbered consecutively. This is important with Byzantine faults to ensure that the primary of a view is not faulty for more than f consecutive views. The mechanism used to select the new primary in Paxos and Viewstamped replication does not have this property.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request and sending this assignment to the backups. But the primary may be faulty: it may assign the same sequence number to different requests, it may stop assigning sequence numbers, or it may leave gaps between request sequence numbers. Therefore, the backups check the sequence numbers assigned by the primary and trigger *view changes* to select a new primary when it appears that the current one has failed.

The remainder of this section describes a simplified version of the algorithm informally. We omit details related to message retransmissions and some important optimizations. These are explained in Chapter 5. We present a formal specification of the algorithm in Section 2.4.4.

2.3.1 Quorums and Certificates

To order requests correctly despite failures, we rely on *quorums* [Gif79]. We could use any Byzantine dissemination quorum system construction [MR97] but currently our quorums are just sets with at least $2f + 1$ replicas. Since there are $3f + 1$ replicas, quorums have two important properties:

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

These properties enable the use of quorums as a reliable memory for protocol information. The information is written to quorums and replicas collect *quorum certificates*, which are sets with one message from each element in a quorum saying that it stored the 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.

2.3.2 The Client

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

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

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

The client waits for a weak certificate with $f + 1$ replies with valid signatures 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 broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

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.

2.3.3 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 2-1 shows the operation of the algorithm in the normal case of no primary faults. Replica 0 is the primary and replica 3 is faulty.

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 2.3.4. The state can be kept in volatile memory; it does not need to be stable.

When the primary p receives a request m from a client, it assigns a sequence number n to m. Then it multicasts a pre-prepare message with the assignment to the backups and inserts this message in its log. The message has the form $\langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_p}$, where v indicates the view in which the message is being sent.

Like pre-prepares, the prepare and commit messages sent in the other phases also contain n

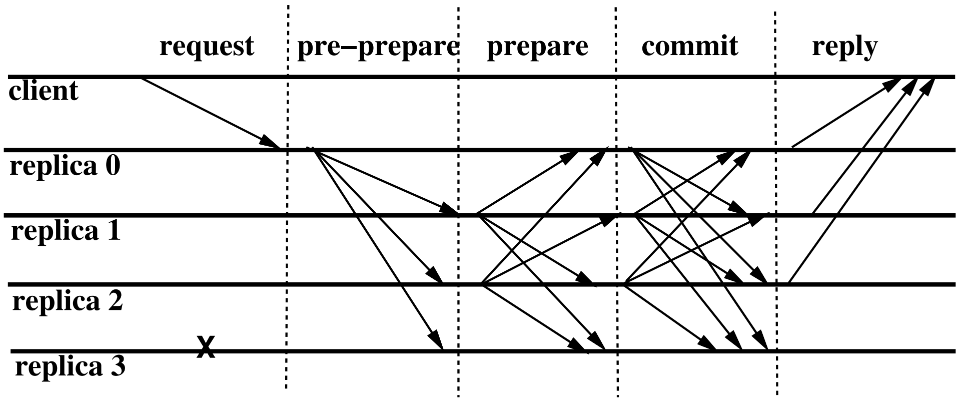

Figure 2-1: Normal Case Operation

and v. A replica only accepts one of these messages provided it is in view v; it can verify the authenticity of the message; and n is between a low water mark, h, and a high water mark, H. The last condition is necessary to enable garbage collection and 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 2.3.4.

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 request. If i accepts the pre-prepare, it enters the *prepare* phase by multicasting a $\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$ message with m's digest d 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. After this point, replicas agree on an order for requests in the same view. 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 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 is not sufficient to ensure a total order for requests across view changes however. Replicas may collect prepared certificates in different views with the same sequence number and different requests. The following example illustrates the problem. A replica collects a prepared certificate in view v for m with sequence number n. The primary for v is faulty and there is a view change. The new primary may not have the prepared certificate. It may even have accepted a pre-prepare message in v for a distinct request with the same sequence number. The new primary may try to prevent conflicting sequence number assignments by reading ordering information from a quorum. It is guaranteed to obtain one reply from a correct replica that assigned n to m in v but it may also receive conflicting replies or replies from replicas that never assigned sequence number n. Unfortunately, there is no way to ensure it will choose the correct one.

The commit phase solves this problem as follows. Each replica i multicasts $\langle \text{COMMIT}, v, n, d, i\rangle_{\sigma_i}$ 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 digest d 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. New primaries ensure information about committed requests is propagated to new views as follows: they read prepared certificates from a quorum and select the sequence number assignments in the certificates for the latest views. Since prepared certificates for the same view never conflict and cannot be forged, this ensures replicas agree on sequence numbers assigned to requests that committed across views.

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

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

2.3.4 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 will 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. We will refer to the states produced by the execution of these requests as *checkpoints* and we will say that a checkpoint with a proof is a *stable checkpoint*.

When replica i produces a checkpoint, it multicasts a $\langle \text{CHECKPOINT}, v, n, d, i\rangle_{\sigma_i}$ 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 5.3 describes how we manage checkpoints and transfer state between replicas efficiently.

Each replica collects messages until it has a weak certificate with $f + 1$ checkpoint messages (including its own) signed by different replicas with the same sequence number n and digest d. This certificate is the proof of correctness for the checkpoint: it proves that at least one correct replica obtained a checkpoint with sequence number n and digest d. We call this certificate the *stable certificate*. 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.

2.3.5 View Changes

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

View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is *waiting* for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v, the backup starts a view change to move the system to view $v + 1$. It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a $\langle \text{VIEW-CHANGE}, v + 1, n, s, \mathcal{C}, \mathcal{P}, i\rangle_{\sigma_i}$ message to all replicas. Here n is the sequence number of the last stable checkpoint s known to i, $\mathcal{C}$ is the stable certificate for that checkpoint, and $\mathcal{P}$ is a set with a prepared certificate for each request that prepared at i with a sequence number greater than n. Figure 2-2 depicts an instance of the view change protocol.

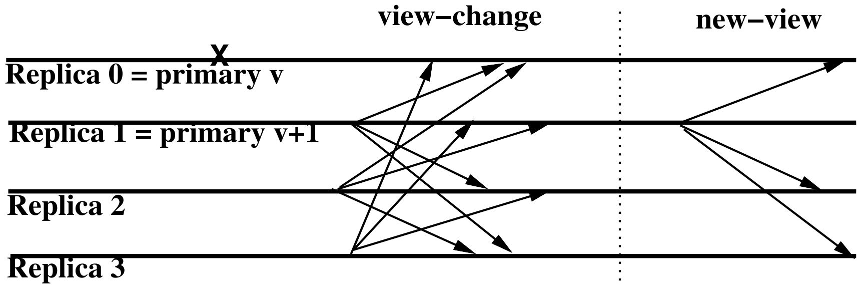

Figure 2-2: View Change Protocol

The new primary p for view $v + 1$ collects a quorum certificate with $2f + 1$ valid view-change messages for view $v + 1$ signed by different replicas (possibly including its own message). We call this certificate the *new-view certificate*. It is guaranteed to contain messages with prepared certificates for all requests that committed in previous views and also for some requests that only prepared. The new primary uses this information to compute a set of pre-prepare messages to send in $v + 1$. This ensures that sequence numbers assigned to committed requests in previous views do not get reassigned to a different request in $v + 1$.

After obtaining a new-view certificate, p multicasts a $\langle \text{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O}, \mathcal{N} \rangle_{\sigma_p}$ message to all other replicas. Here $\mathcal{V}$ is the new-view certificate, and $\mathcal{O} \cup \mathcal{N}$ is the set of pre-prepare messages that propagate sequence number assignments from previous views. $\mathcal{O}$ and $\mathcal{N}$ are computed as follows:

1. The primary determines the sequence number h of the latest stable checkpoint in $\mathcal{V}$ and the highest sequence number H in a prepared certificate in a message in $\mathcal{V}$.
2. The primary creates a new pre-prepare message for view $v + 1$ for each sequence number n such that $h < n \leq H$. There are two cases: (1) there is a prepared certificate in a message in $\mathcal{V}$ with sequence number n, or (2) there is no prepared certificate. In the first case, the primary adds a new message $\langle \text{PRE-PREPARE}, v+1, n, m \rangle_{\sigma_p}$ to $\mathcal{O}$, where m is the request in a prepared certificate with sequence number n and with the highest view number in a message in $\mathcal{V}$. In the second case, it adds a new pre-prepare message $\langle \text{PRE-PREPARE}, v+1, n, \mathit{null} \rangle_{\sigma_p}$ to $\mathcal{N}$. Here, *null* is the digest of a special null request; a null request goes through the protocol like other requests, but its execution is a no-op. (Paxos [Lam89] used a similar technique to fill in gaps.)

Next the primary appends the messages in $\mathcal{O}$ and $\mathcal{N}$ to its log. If h is greater than the sequence number of its latest stable checkpoint, the primary also adds the stable certificate for the checkpoint with sequence number h to its log and discards information from the log as discussed in Section 2.3.4. If h is greater than the primary's current state, it also updates its current state to be equal to the checkpoint with sequence number h. Then it *enters* view $v + 1$: at this point it is able to accept messages for view $v + 1$.

A backup accepts a new-view message for view $v+1$ if it is signed properly, if it contains a valid new-view certificate for view $v+1$, and if the sets $\mathcal{O}$ and $\mathcal{N}$ are correct: it verifies the correctness of these sets by performing a computation similar to the one used by the primary to create them. These checks prevent backups from accepting sequence number assignments that conflict with requests that committed in previous views. Then the backup adds the new information to its log as described for the primary, multicasts a prepare for each message in $\mathcal{O} \cup \mathcal{N}$ to all the other replicas, adds these prepares to its log, and enters view $v + 1$.

Thereafter, the protocol proceeds as described in Section 2.3.3. Replicas redo the protocol for messages between h and H but they avoid re-executing client requests by using their stored information about the last reply sent to each client.

Liveness

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

First, to avoid starting a view change too soon, a replica that multicasts a view-change message for view $v + 1$ waits for $2f + 1$ view-change messages for view $v + 1$ 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 $p = v \ \mathit{mod}\ |\mathcal{R}|$, the primary cannot be faulty for more than f consecutive views.

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

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.

2.4 Formal Model

This section presents a formalization of BFT-PK using I/O automata [Lyn96]. It starts with a brief introduction to I/O automata. Then, it presents a formal description of the system model and assumptions behind BFT-PK. Section 2.4.3 provides a specification for the modified linearizability condition implemented by BFT-PK and Section 2.4.4 contains the specification for the algorithm ran by clients and replicas. We present a formal safety proof for BFT-PK in Appendix A.

2.4.1 I/O Automata

An I/O automaton is an automaton with (possibly infinite) *state* and with an *action* labeling each transition. These actions have a *pre-condition*, which determines whether they are *enabled*, and they have *effects*, which determine how the state is modified when they execute. 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: first, an enabled action is selected non-deterministically, and then it is executed. Several automata can be composed by combining input and output actions. Lynch's book [Lyn96] provides a good description of I/O automata.

2.4.2 System Model

The algorithm can replicate any service that can be modeled by a deterministic state machine as defined in Definition 2.4.1. The requirement that the state machine's transition function g be total means that the service behavior must be well defined for all possible operations and arguments. This is important to ensure non-faulty replicas produce the same results even when they are requested to execute invalid operations. 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.

Definition 2.4.1 *A deterministic state machine is a tuple $\langle \mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{O}', g, s_o \rangle$. It has a state in a set $\mathcal{S}$ (initially equal to s_o) and its behavior is defined by a transition function:*

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

The arguments to the function are a client identifier in $\mathcal{C}$, 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; g must be total.

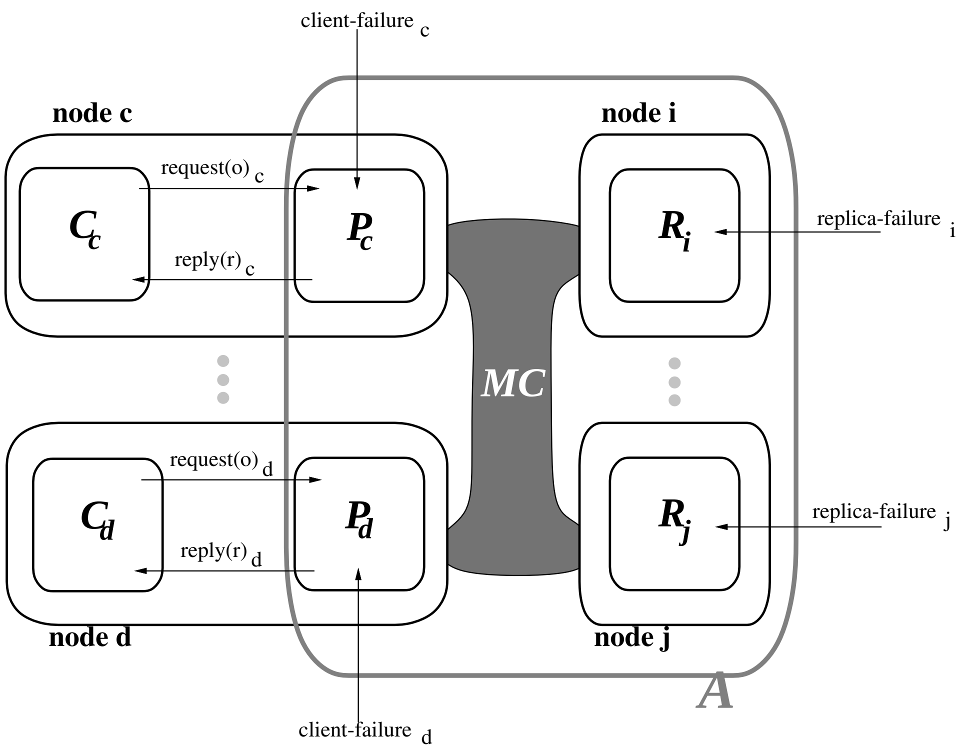

Figure 2-3: System Model

The distributed system that implements a replicated state machine $\langle \mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{O}', g, s_o \rangle$ is modeled as a set of I/O automata [Lyn96]. Each client has a unique identifier c in $\mathcal{C}$ and is modeled by a client automaton C_c. The composition of all clients is denoted by C. The replicated service is modeled as an automaton A that is the composition of three types of automata: proxy, multicast channel, and replica. Figure 2-3 shows the architecture of the system and Figure 2-4 presents the external interface of A.

Input:	$\text{REQUEST}(o)_c$, $o \in \mathcal{O}$, $c \in \mathcal{C}$
	CLIENT-FAILURE_c, $c \in \mathcal{C}$
	REPLICA-FAILURE_i, $i \in \mathcal{R}$
Output:	$\text{REPLY}(r)_c$, $r \in \mathcal{O}'$, $c \in \mathcal{C}$

Figure 2-4: External Signature of the Replicated Service Automaton, A

There is a proxy automaton P_c for each client C_c. P_c provides an input action for client 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, REPLY$(r)_c$. The communication between C_c and P_c does not involve any network; they are assumed to execute in the same node in the distributed system. P_c communicates with a set of state machine 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. Automata have no access to the state components of automata running on other nodes in the distributed system. They communicate through an unreliable network.

Signature:
Input: SEND$(m, X)_x$
Internal: MISBEHAVE(m, X, X')
Output: RECEIVE$(m)_x$

Here, $m \in \mathcal{M}$, $X, X' \subseteq \mathcal{X}$, and $x \in \mathcal{X}$

State:
wire $\subseteq \mathcal{M} \times 2^{\mathcal{X}}$, initially {}

Transitions:

SEND$(m, X)_x$
Eff: $\textit{wire} := \textit{wire} \cup \{(m, X)\}$

RECEIVE$(m)_x$
Pre: $\exists (m, X) \in \textit{wire} : (x \in X$
Eff: $\textit{wire} := \textit{wire} - \{(m, X)\} \cup \{(m, X - \{x\})\})$

MISBEHAVE(m, X, X')
Pre: $(m, X) \in \textit{wire}$
Eff: $\textit{wire} := \textit{wire} - \{(m, X)\} \cup \{(m, X')\}$

Figure 2-5: Network specification: multicast channel automaton.

The network between replicas and proxies is modeled as the multicast channel automaton, MC, defined in Figure 2-5. There is a single multicast automaton in the system with SEND and RECEIVE actions for each proxy and replica. These actions allow automata to send messages in a universal message set $\mathcal{M}$ to any subset of automata with identifiers in $\mathcal{X} = \mathcal{C} \cup \mathcal{R}$. The automaton has a single state component *wire* that stores pairs with a message and a destination set. It does not provide authenticated communication; the RECEIVE actions do not identify the sender of the message.

The SEND actions simply add the argument message and its destination set to *wire* and the RECEIVE actions deliver a message to one of the elements in its destination set (and remove this element from the set). The MISBEHAVE actions allow the channel to lose messages or duplicate them and the RECEIVE actions are defined such that messages may be reordered. Additionally, the automaton is defined such that every message that was ever sent on the channel is remembered in *wire*. This allows the MISBEHAVE actions to simulate replays of any of these messages by an attacker. We do not assume synchrony. The nodes are part of an asynchronous distributed system with no known bounds on message delays or on the time for automata to take enabled actions.

We use a Byzantine failure model, i.e., faulty clients and replicas may behave arbitrarily (except for the restrictions discussed next). The CLIENT-FAILURE and REPLICA-FAILURE actions are used to model client and replica failures. Once such a failure action occurs the corresponding automaton is replaced by an arbitrary automaton with the same external interface and it remains faulty for the rest of the execution. We assume however that this arbitrary automaton has a state component called *faulty* that is set to true. It is important to understand that the failure actions and the *faulty* variables are used only to model failures formally for the correctness proof; our algorithm does not know whether a client or replica is faulty or not.

As discussed in Section 2.1, the algorithm uses digital signatures and cryptographic hash functions. We assume the signature scheme is non-existentially forgeable even with an adaptive chosen message attack [GMR88] and that the cryptographic hash function is collision resistant [Dam89]. These assumptions amount to restrictions on the computational power of the adversary and the Byzantine-faulty replicas and clients it may control.

2.4.3 Modified Linearizability

The safety property offered by BFT-PK is a form of linearizability [HW87]: the replicated service behaves like 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 REQUEST and REPLY interface provided by the replicated service automaton. 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 non-faulty clients differently.

A similar modification to linearizability was proposed concurrently in [MRL98]. Their proposal uses conditions on execution traces to specify the modified linearizability property. We specify the property using an I/O automaton, S, with the same external signature as the replicated service automaton, A. Our approach has several advantages: it produces a simpler specification and it enables the use of state-based proof techniques like invariant assertions and simulation relations to reason about linearizability. These proof 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, S, is a simple, abstract, centralized implementation of the state machine $\langle \mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{O}', g, s_o \rangle$ that is defined in Figure 2-6. We say that A satisfies the safety property if it implements S.

The state of S 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 $\textit{last-req}_c$ component is used to timestamp requests by client c to totally order them, and $\textit{last-rep-t}_c$ remembers the value of $\textit{last-req}_c$ that was associated with the last operation executed for c. The $\textit{faulty-client}_c$ and $\textit{faulty-replica}_i$ indicate which clients and replicas are faulty.

The CLIENT-FAILURE and REPLICA-FAILURE actions are used to model failures; they set the $\textit{faulty-client}_c$ or the $\textit{faulty-replica}_i$ variables to true. The $\text{REQUEST}(o)_c$ actions increment $\textit{last-req}_c$

Signature:

Input: $\text{REQUEST}(o)_c$
CLIENT-FAILURE_c
REPLICA-FAILURE_i

Internal: $\text{EXECUTE}(o, t, c)$
$\text{FAULTY-REQUEST}(o, t, c)$

Output: $\text{REPLY}(r)_c$

Here, $o \in \mathcal{O}$, $t \in \mathbf{N}$, $c \in \mathcal{C}$, $i \in \mathcal{R}$, and $r \in \mathcal{O}'$

State:

$val \in \mathcal{S}$, initially s_o
$in \subseteq \mathcal{O} \times \mathbf{N} \times \mathcal{C}$, initially $\{\}$
$out \subseteq \mathcal{O}' \times \mathbf{N} \times \mathcal{C}$, initially $\{\}$
$\forall c \in \mathcal{C}$, $\textit{last-req}_c \in \mathbf{N}$, initially $\textit{last-req}_c = 0$
$\forall c \in \mathcal{C}$, $\textit{last-rep-t}_c \in \mathbf{N}$, initially $\textit{last-rep-t}_c = 0$
$\forall c \in \mathcal{C}$, $\textit{faulty-client}_c \in \textit{Bool}$, initially $\textit{faulty-client}_c = \textit{false}$
$\forall i \in \mathcal{R}$, $\textit{faulty-replica}_i \in \textit{Bool}$, initially $\textit{faulty-replica}_i = \textit{false}$
$\textit{n-faulty} \equiv |\{ i \mid \textit{faulty-replica}_i = \textit{true} \}|$

Transitions (if $\textit{n-faulty} \leq \lfloor \frac{|\mathcal{R}|-1}{3} \rfloor$**):**

$\text{REQUEST}(o)_c$
Eff: $\textit{last-req}_c := \textit{last-req}_c + 1$
$in := in \cup \{\langle o, \textit{last-req}_c, c\rangle\}$

CLIENT-FAILURE_c
Eff: $\textit{faulty-client}_c := \textit{true}$

REPLICA-FAILURE_i
Eff: $\textit{faulty-replica}_i := \textit{true}$

$\text{REPLY}(r)_c$
Pre: $\textit{faulty-client}_c = \textit{true} \vee \exists t : (\langle r, t, c\rangle \in out$
Eff: $out := out - \{\langle r, t, c\rangle\})$

$\text{FAULTY-REQUEST}(o, t, c)$
Pre: $\textit{faulty-client}_c = \textit{true}$
Eff: $in := in \cup \{\langle o, t, c\rangle\}$

$\text{EXECUTE}(o, t, c)$
Pre: $\langle o, t, c\rangle \in in$
Eff: $in := in - \{\langle o, t, c\rangle\}$
if $t > \textit{last-rep-t}_c$ then
$(r, val) := g(c, o, val)$
$out := out \cup \{\langle r, t, c\rangle\}$
$\textit{last-rep-t}_c := t$

Figure 2-6: Specification of Safe Behavior, S

to obtain a new timestamp for the request, and add a triple to *in* with the requested operation, o, the timestamp value, $\textit{last-req}_c$, and the client identifier. The FAULTY-REQUEST actions are similar. They model execution of requests by faulty clients that bypass the external signature of A, e.g., 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 non-faulty 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 REPLY precondition is weaker for faulty clients to allow arbitrary replies for such clients. The algorithm cannot guarantee safety if more than $\lfloor \frac{|\mathcal{R}|-1}{3} \rfloor$ replicas are faulty. Therefore, the behavior of S is left unspecified in this case.

2.4.4 Algorithm Specification

Proxy. Each client C_c interacts with the replicated service through a proxy automaton P_c, which is defined in Figure 2-7.

Signature:

Input: $\text{REQUEST}(o)_c$
$\text{RECEIVE}(\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i})_c$
CLIENT-FAILURE_c

Output: $\text{REPLY}(r)_c$
$\text{SEND}(m, X)_c$

Here, $o \in \mathcal{O}$, $v, t \in \mathbf{N}$, $c \in \mathcal{C}$, $i \in \mathcal{R}$, $r \in \mathcal{O}'$, $m \in M$, and $X \subseteq \mathcal{X}$

State:

$\textit{view}_c \in \mathbf{N}$, initially 0
$\textit{in}_c \subseteq \mathcal{M}$, initially $\{\}$
$\textit{out}_c \subseteq \mathcal{M}$, initially $\{\}$
$\textit{last-req}_c \in \mathbf{N}$, initially 0
$\textit{retrans}_c \in \textit{Bool}$, initially *false*
$\textit{faulty}_c \in \textit{Bool}$, initially *false*

Transitions:

$\text{REQUEST}(o)_c$
Eff: $\textit{last-req}_c := \textit{last-req}_c + 1$
$\textit{out}_c := \{\langle \text{REQUEST}, o, \textit{last-req}_c, c \rangle_{\sigma_c}\}$
$\textit{in}_c := \{\}$
$\textit{retrans}_c := \textit{false}$

$\text{RECEIVE}(\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i})_c$
Eff: if $(\textit{out}_c \neq \{\} \wedge \textit{last-req}_c = t)$ then
$\textit{in}_c := \textit{in}_c \cup \{\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}\}$

CLIENT-FAILURE_c
Eff: $\textit{faulty}_c := \textit{true}$

$\text{SEND}(m, \{\textit{view}_c \bmod |\mathcal{R}|\})_c$
Pre: $m \in \textit{out}_c \wedge \neg \textit{retrans}_c$
Eff: $\textit{retrans}_c := \textit{true}$

$\text{SEND}(m, \mathcal{R})_c$
Pre: $m \in \textit{out}_c \wedge \textit{retrans}_c$
Eff: none

$\text{REPLY}(r)_c$
Pre: $\textit{out}_c \neq \{\} \wedge \exists R : (|R| > f \wedge \forall i \in R : (\exists v : (\langle \text{REPLY}, v, \textit{last-req}_c, c, i, r \rangle_{\sigma_i} \in \textit{in}_c)))$
Eff: $\textit{view}_c := \max(\{v | \langle \text{REPLY}, v, \textit{last-req}_c, c, i, r \rangle_{\sigma_i} \in \textit{in}_c\})$
$\textit{out}_c := \{\}$

Figure 2-7: Proxy automaton

The proxy remembers the last request sent to the replicas in $\textit{out}_c$ and it collects replies that match this request in $\textit{in}_c$. It uses $\textit{last-req}_c$ to generate timestamps for requests, $\textit{view}_c$ to track the current view of the replicated system, and $\textit{retrans}_c$ to indicate whether a request is being retransmitted.

The REQUEST actions add a request for the argument operation to $\textit{out}_c$. This request is sent on the multicast channel when one of the SEND actions execute: requests are sent first to the primary of $view_c$ and are retransmitted to all replicas. The RECEIVE actions collect replies in in_c that match the request in out_c. Once there are more than f replies in in_c, the REPLY action becomes enabled and returns the result of the requested operation to the client.

Replica. The signature and state of replica automata are described in Figure 2-8.

Signature:

Input: $\text{RECEIVE}(\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})_i$
$\text{RECEIVE}(\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j})_i$
$\text{RECEIVE}(\langle \text{PREPARE}, v, n, d, j\rangle_{\sigma_j})_i$
$\text{RECEIVE}(\langle \text{COMMIT}, v, n, d, j\rangle_{\sigma_j})_i$
$\text{RECEIVE}(\langle \text{CHECKPOINT}, v, n, d, j\rangle_{\sigma_j})_i$
$\text{RECEIVE}(\langle \text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j})$
$\text{RECEIVE}(\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$
REPLICA-FAILURE_i

Internal: $\text{SEND-PRE-PREPARE}(m, v, n)_i$
$\text{SEND-COMMIT}(m, v, n)_i$
$\text{EXECUTE}(m, v, n)_i$
$\text{VIEW-CHANGE}(v)_i$
$\text{SEND-NEW-VIEW}(v, V)_i$
COLLECT-GARBAGE_i

Output: $\text{SEND}(m, X)_c$

Here, $t, v, n \in \mathbf{N}$, $c \in \mathcal{C}$, $i, j \in \mathcal{R}$, $m \in \mathcal{M}$, $s \in \mathcal{V}'$, $V, O, N, C, P \subseteq \mathcal{M}$, $X \subseteq \mathcal{X}$, and $d \in \mathcal{D}'$
where $\mathcal{V}' = \mathcal{V} \times (\mathcal{C} \to \mathcal{O}) \times (\mathcal{C} \to \mathbf{N})$ and $\mathcal{D}' = \{d \mid \exists s \in \mathcal{V}' : (d = D(s))\}$

State:

$val_i \in \mathcal{S}$, initially s_o
$\textit{last-rep}_i : \mathcal{C} \to \mathcal{O}'$, initially $\forall c \in \mathcal{C} : \textit{last-rep}_i(c) = \textit{null-rep}$
$\textit{last-rep-t}_i : \mathcal{C} \to \mathbf{N}$, initially $\forall c \in \mathcal{C} : \textit{last-rep-t}_i(c) = 0$
$\textit{chkpts}_i \subseteq \mathbf{N} \times \mathcal{V}'$, initially $\{\langle 0, \langle v_0, \textit{null-rep}, 0\rangle\rangle\}$
$in_i \subseteq \mathcal{M}$, initially $\{\langle \text{CHECKPOINT}, 0, D(\langle v_0, \textit{null-rep}, 0\rangle), k\rangle_{\sigma_k} \mid \forall k \in \mathcal{R}\}$
$out_i \subseteq \mathcal{M}$, initially $\{\}$
$view_i \in \mathbf{N}$, initially 0
$\textit{last-exec}_i \in \mathbf{N}$, initially 0
$seqno_i \in \mathbf{N}$, initially 0
$faulty_i \in Bool$, initially *false*
$h_i \equiv min(\{n \mid \langle n, \nu\rangle \in \textit{chkpts}_i\})$
$\textit{stable-chkpt}_i \equiv \nu \mid \langle h_i, \nu\rangle \in \textit{chkpts}_i$

Figure 2-8: Signature and State of Replica Automaton i

The state variables of the automaton for replica i include the current value of the replica's copy of the state machine, val_i, the last reply $\textit{last-rep}_i$ sent to each client, and the timestamps in those replies $\textit{last-rep-t}_i$. There is also a set of checkpoints, $\textit{chkpts}_i$, whose elements contain not only a snapshot of val_i but also a snapshot of $\textit{last-rep}_i$ and $\textit{last-rep-t}_i$. The log with messages received or sent by the replica is stored in in_i and out_i buffers messages that are about to be sent on the multicast channel. Replica's also maintain the current view number, $view_i$, the sequence number of the last request executed, $last\text{-}exec_i$, and, if they are the primary, the sequence number assigned to the last request, $seqno_i$.

$$tag(m,u) \equiv m = \langle u, ... \rangle$$
$$primary(v) \equiv v \bmod |\mathcal{R}|$$
$$primary(i) \equiv view_i \bmod |\mathcal{R}|$$
$$in\text{-}v(v,i) \equiv view_i = v$$
$$in\text{-}w(n,i) \equiv 0 < n - h_i \leq L, \text{ where } L \in \mathbf{N}$$
$$in\text{-}wv(v,n,i) \equiv in\text{-}w(n,i) \wedge in\text{-}v(v,i)$$
$$prepared(m,v,n,M) \equiv \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_{primary(v)}} \in M \wedge$$
$$\quad \exists R : (|R| \geq 2f \wedge primary(v) \notin R \wedge \forall k \in R : (\langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} \in M))$$
$$prepared(m,v,n,i) \equiv prepared(m,v,n,in_i)$$
$$last\text{-}prepared(m,v,n,M) \equiv prepared(m,v,n,M) \wedge$$
$$\quad \nexists m', v' : ((prepared(m',v',n,M) \wedge v' > v) \vee (prepared(m',v,n,M) \wedge m \neq m'))$$
$$last\text{-}prepared(m,v,n,i) \equiv last\text{-}prepared(m,v,n,in_i)$$
$$committed(m,v,n,i) \equiv (\exists v' : (\langle \text{PRE-PREPARE}, v', n, m\rangle_{\sigma_{primary(v')}} \in in_i) \vee m \in in_i) \wedge$$
$$\quad \exists R : (|R| \geq 2f+1 \wedge \forall k \in R : (\langle \text{COMMIT}, v, n, D(m), k\rangle_{\sigma_k} \in in_i))$$
$$correct\text{-}view\text{-}change(m,v,j) \equiv \exists n,s,C,P : (m = \langle \text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j} \wedge$$
$$\quad \exists R : (|R| > f \wedge \forall k \in R : (\exists v'' < v : (\langle \text{CHECKPOINT}, v'', n, D(s), k\rangle_{\sigma_k} \in C)) \wedge$$
$$\quad \forall \langle \text{PRE-PREPARE}, v', n', m'\rangle_{\sigma_{primary(v')}} \in P :$$
$$\qquad (last\text{-}prepared(m',v',n',P) \wedge v' < v \wedge 0 < n' - n \leq L)$$
$$merge\text{-}P(V) \equiv \{\, m \mid \exists \langle \text{VIEW-CHANGE}, v, n, s, C, P, k\rangle_{\sigma_k} \in V : (m \in P) \,\}$$
$$max\text{-}n(M) \equiv max(\{\, n \mid \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i} \in M \vee \langle \text{VIEW-CHANGE}, v, n, s, C, P, i\rangle_{\sigma_i} \in M\})$$
$$correct\text{-}new\text{-}view(m,v) \equiv$$
$$\quad \exists V,O,N,R : (m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_{primary(v)}} \wedge |V| = |R| = 2f+1 \wedge$$
$$\quad \forall k \in R : (\exists m' \in V : (correct\text{-}view\text{-}change(m',v,k))) \wedge$$
$$\quad O = \{\, \langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_{primary(v)}} \mid n > max\text{-}n(V) \wedge \exists v' : last\text{-}prepared(m',v',n,merge\text{-}P(V))\} \wedge$$
$$\quad N = \{\, \langle \text{PRE-PREPARE}, v, n, null\rangle_{\sigma_{primary(v)}} \mid max\text{-}n(V) < n < max\text{-}n(O) \wedge$$
$$\qquad \nexists v', m', n : last\text{-}prepared(m',v',n,merge\text{-}P(V)))$$
$$update\text{-}state\text{-}nv(i,v,V,m) \equiv$$
$$\quad \text{if } max\text{-}n(V) > h_i \text{ then}$$
$$\qquad in_i := in_i \cup (\text{pick } C : \exists \langle \text{VIEW-CHANGE}, v, max\text{-}n(V), s, C, P, k\rangle_{\sigma_k} \in V)$$
$$\qquad \text{if } \langle \text{CHECKPOINT}, v, max\text{-}n(V), D(s), i\rangle_{\sigma_i} \notin in_i \text{ then}$$
$$\qquad\quad in_i = in_i \cup \{\langle \text{CHECKPOINT}, v, max\text{-}n(V), D(s), i\rangle_{\sigma_i}\}$$
$$\qquad\quad out_i = out_i \cup \{\langle \text{CHECKPOINT}, v, max\text{-}n(V), D(s), i\rangle_{\sigma_i}\}$$
$$\qquad chkpts_i := chkpts_i - \{p = \langle n', s'\rangle \mid p \in chkpts_i \wedge n' < max\text{-}n(V)\}$$
$$\qquad \text{if } max\text{-}n(V) > last\text{-}exec_i \text{ then}$$
$$\qquad\quad chkpts_i := chkpts_i \cup \{\langle max\text{-}n(V), s\rangle \mid \exists \langle \text{VIEW-CHANGE}, v, max\text{-}n(V), s, C, P, k\rangle_{\sigma_k} \in V\}$$
$$\qquad\quad (val_i, last\text{-}rep_i, last\text{-}rep\text{-}t_i) := stable\text{-}chkpt_i$$
$$\qquad\quad last\text{-}exec_i := max\text{-}n(V)$$
$$has\text{-}new\text{-}view(v,i) \equiv v = 0 \vee \exists m : (m \in in_i \wedge correct\text{-}new\text{-}view(m,v))$$
$$take\text{-}chkpt(n) \equiv (n \bmod chkpt\text{-}int) = 0, \text{ where } chkpt\text{-}int \in \mathbf{N} \wedge chkpt\text{-}int < L$$

Figure 2-9: Auxiliary Functions

Figure 2-9 defines several auxiliary functions that are used in the specification of replicas' actions. The $tag(m,u)$ predicate is true if and only if the tag of message m is u. The function $primary(v)$ returns the identifier of the primary replica for view v and $primary(i)$ returns the identifier of the primary for the view with number $view_i$.

The next three predicates are used by replicas to decide which messages to log: $in\text{-}v(v,i)$ is true if and only if v equals i's current view; *in-w*(n, i) is true if and only if sequence number n is between the low and high water marks in i's log; and *in-wv*(v, n, i) is the conjunction of the two.

The *prepared*(m, v, n, M) predicate is true if and only if there is a prepared certificate in M for request m with sequence number n and view v. *last-prepared*(m, v, n, M) is true if and only if the certificate with view v is the one with the greatest view number for sequence number n. The predicate *committed*(m, v, n, i) is true provided the request is committed at replica i: there is a committed certificate in in_i for request m with sequence number n and view v, and m (or a pre-prepare message containing m) is also in in_i.

The *correct-view-change*(m, v, j) and *correct-new-view*(m, v) predicates check the correctness of view-change and new-view messages, respectively. The function *update-state-nv* updates the replica's checkpoints and current state after receiving (or sending) a new-view message. Section 2.3.5 explains how correct view-change and new-view messages are built and how the state is updated. Finally, *has-new-view*(v, i) returns true if replica i is in view 0 or has a valid new-view message for view v, and *take-chkpt*(n) returns true if n is the sequence number of a checkpoint (as explained in Section 2.3.4).

$\text{SEND}(m, \mathcal{R} - \{i\})_i$
  Pre: $m \in out_i \wedge \neg tag(m, \text{REQUEST}) \wedge \neg tag(m, \text{REPLY})$
  Eff: $out_i := out_i - \{m\}$

$\text{SEND}(m, \{primary(i)\})_i$
  Pre: $m \in out_i \wedge tag(m, \text{REQUEST})$
  Eff: $out_i := out_i - \{m\}$

$\text{SEND}(\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}, \{c\})_i$
  Pre: $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i} \in out_i$
  Eff: $out_i := out_i - \{\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}\}$

Figure 2-10: Output Actions

The replica's output actions are defined in Figure 2-10. They are very simple: actions of the first type multicast messages to the other replicas, the others are used to forward requests to the primary and to send replies to the clients, respectively. Figure 2-11 presents the garbage collection actions, which are also simple. The RECEIVE actions collect checkpoint messages in the log and the COLLECT-GARBAGE actions discard old messages and checkpoints when the replica has a stable certificate logged.

Figure 2-12 presents the actions associated with the normal-case protocol. The actions match the description in Section 2.3.3 closely but there are some details that were omitted in that description. For example, pre-prepare messages are sent by the primary or accepted by the backups only if the replica has a new-view message logged for its current view; this is important to ensure the replica has enough information to prevent conflicting sequence number assignments.

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 *take-chkpt* is true, the replica takes a checkpoint by adding a snapshot of of val_i, $last\text{-}rep_i$, and $last\text{-}rep\text{-}t_i$ to the checkpoint set and puts a matching checkpoint message in out_i to be multicast to the other replicas.

$\text{RECEIVE}(\langle \text{CHECKPOINT}, v, n, d, j\rangle_{\sigma_j})_i \ (j \neq i)$
  Eff: if $view_i \geq v \wedge in\text{-}w(n, i)$ then
    $in_i := in_i \cup \{\langle \text{CHECKPOINT}, v, n, d, j\rangle_{\sigma_j}\}$

COLLECT-GARBAGE_i
  Pre: $\exists R, n, d : (|R| > f \wedge i \in R \ \wedge \forall k \in R : (\exists v : (\langle \text{CHECKPOINT}, v, n, d, k\rangle_{\sigma_k} in_i)$
  Eff: $in_i := in_i - \{m = \langle \text{PRE-PREPARE}, v', n', m'\rangle_{\sigma_j} | m \in in_i \wedge n' \leq n\}$
    $in_i := in_i - \{m = \langle \text{PREPARE}, v', n', d', j\rangle_{\sigma_j} | m \in in_i \wedge n' \leq n\}$
    $in_i := in_i - \{m = \langle \text{COMMIT}, v', n', d', j\rangle_{\sigma_j} | m \in in_i \wedge n' \leq n\}$
    $in_i := in_i - \{m = \langle \text{CHECKPOINT}, v', n', d', j\rangle_{\sigma_j} | m \in in_i \wedge n' < n\}$
    $chkpts_i := chkpts_i - \{p = \langle n', s\rangle | p \in chkpts_i \wedge n' < n\})$

Figure 2-11: Garbage Collection Actions

The last set of actions is presented in Figure 2-13. These actions define the behavior of the replica automata during view changes and are more complex. The SEND-VIEW-CHANGE action increments the view number and builds a new view-change message that is put in out_i to be multicast to the other replicas. This view-change message contains the replica's stable checkpoint sequence number, h_i, the stable checkpoint, $stable\text{-}chkpt_i$, a copy of the stable certificate in the replica's log, C, and a copy of the prepared certificates in the log with the highest view number for each sequence number. The replicas collect view-change messages that are correct and have a view number greater than or equal to their current view.

The SEND-NEW-VIEW$(v, V)_i$ action is enabled when the new primary has a new-view certificate, V, in the log for view v. When this action executes, the primary picks the checkpoint with the highest sequence number, $h = max\text{-}n(V)$, to be the start state for request processing in the new view. Then it computes the sets O and N with pre-prepare messages for view v: O has a message for each request with a prepared certificate in some message in V with sequence number greater than h; and N has a pre-prepare for the null request for every sequence number between $max\text{-}n(V)$ and $max\text{-}n(O)$ without a message in O. The new-view message includes V, N, and O. The new primary updates $seqno_i$ to be $max\text{-}n(O)$ to ensure it will not assign sequence numbers that are already assigned in O. If needed, the *update-state-nv* function updates the replica's checkpoint set and val_i to reflect the information in V.

When the backups receive the new-view message, they check if it is correct. If it is, they update their state like the primary and they add prepare messages for each message in $O \cup N$ to out_i to be multicast to the other replicas.

RECEIVE($\langle$REQUEST$, o, t, c\rangle_{\sigma_c})_i$
  Eff: let $m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c}$
    if $t = \textit{last-rep-t}_i(c)$ then
      $\textit{out}_i := \textit{out}_i \cup \{\langle$REPLY$, \textit{view}_i, t, c, i, \textit{last-rep}_i(c)\rangle_{\sigma_i}\}$
    else if $t > \textit{last-rep-t}_i(c)$ then
      $\textit{in}_i := \textit{in}_i \cup \{m\}$
      if $\textit{primary}(i) \neq i$ then
        $\textit{out}_i := \textit{out}_i \cup \{m\}$

SEND-PRE-PREPARE$(m, v, n)_i$
  Pre: $\textit{primary}(i) = i \wedge \textit{seqno}_i = n - 1 \wedge \textit{in-wv}(v, n, i) \wedge \textit{has-new-view}(v, i) \wedge$
    $\exists o, t, c : (m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c} \wedge m \in \textit{in}_i) \wedge \nexists \langle$PRE-PREPARE$, v, n', m\rangle_{\sigma_i} \in \textit{in}_i$
  Eff: $\textit{seqno}_i := \textit{seqno}_i + 1$
    let $p = \langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_i}$
      $\textit{out}_i := \textit{out}_i \cup \{p\}$
      $\textit{in}_i := \textit{in}_i \cup \{p\}$

RECEIVE$\langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_j})_i$ $(j \neq i)$
  Eff: if $j = \textit{primary}(i) \wedge \textit{in-wv}(v, n, i) \wedge \textit{has-new-view}(v, i) \wedge$
    $\nexists d : (d \neq D(m) \wedge \langle$PREPARE$, v, n, d, i\rangle_{\sigma_i} \in \textit{in}_i)$ then
    let $p = \langle$PREPARE$, v, n, D(m), i\rangle_{\sigma_i}$
      $\textit{in}_i := \textit{in}_i \cup \{\langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_j}, p\}$
      $\textit{out}_i := \textit{out}_i \cup \{p\}$

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

SEND-COMMIT$(m, v, n)_i$
  Pre: $\textit{prepared}(m, v, n, i) \wedge \langle$COMMIT$, v, n, D(m), i\rangle_{\sigma_i} \notin \textit{in}_i$
  Eff: let $c = \langle$COMMIT$, v, n, D(m), i\rangle_{\sigma_i}$
    $\textit{out}_i := \textit{out}_i \cup \{c\}$
    $\textit{in}_i := \textit{in}_i \cup \{c\}$

RECEIVE($\langle$COMMIT$, v, n, d, j\rangle_{\sigma_j})_i$ $(j \neq i)$
  Eff: if $\textit{view}_i \geq v \wedge \textit{in-w}(n, i)$ then
    $\textit{in}_i := \textit{in}_i \cup \{\langle$COMMIT$, v, n, d, j\rangle_{\sigma_j}\}$

EXECUTE$(m, v, n)_i$
  Pre: $n = \textit{last-exec}_i + 1 \wedge \textit{committed}(m, v, n, i)$
  Eff: $\textit{last-exec}_i := n$
    if $(m \neq \textit{null})$ then
      if $\exists o, t, c : (m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c}$ then
        if $t \geq \textit{last-rep-t}_i(c)$ then
          if $t > \textit{last-rep-t}_i(c)$ then
            $\textit{last-rep-t}_i(c) := t$
            $(\textit{last-rep}_i(c), \textit{val}_i) := g(c, o, \textit{val}_i)$
          $\textit{out}_i := \textit{out}_i \cup \{\langle$REPLY$, \textit{view}_i, t, c, i, \textit{last-rep}_i(c)\rangle_{\sigma_i}\})$
    $\textit{in}_i := \textit{in}_i - \{m\}$
    if $\textit{take-chkpt}(n)$ then
      let $m' = \langle$CHECKPOINT$, \textit{view}_i, n, D(\langle \textit{val}_i, \textit{last-rep}_i, \textit{last-rep-t}_i\rangle), i\rangle_{\sigma_i}$
        $\textit{out}_i := \textit{out}_i \cup \{m'\}$
        $\textit{in}_i := \textit{in}_i \cup \{m'\}$
      $\textit{chkpts}_i := \textit{chkpts}_i \cup \{\langle n, \langle \textit{val}_i, \textit{last-rep}_i, \textit{last-rep-t}_i\rangle\rangle\}$

Figure 2-12: Normal Case Actions

We omitted some details in order to simplify the definitions. For example, we omitted the automata code to ensure fairness, the safe guards to ensure the log size is bounded, and retransmissions. This was done after careful reasoning that adding these details would not affect safety. The other thing we omitted was the automata code to manipulate view-change timers and ensure liveness. Adding this code would not affect safety because it simply adds restrictions to the pre-condition of SEND-VIEW-CHANGE.

REPLICA-FAILURE$_i$
Eff: $faulty_i := true$

SEND-VIEW-CHANGE$(v)_i$
Pre: $v = view_i + 1$
Eff: $view_i := v$
let $P' = \{\langle m, v, n\rangle | last\text{-}prepared(m, v, n, i)\}$,
$P = \bigcup_{\langle m,v,n\rangle \in P'} (\{p = \langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} | p \in in_i\} \cup \{\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma primary(v)}\})$,
$C = \{m' = \langle \text{CHECKPOINT}, v'', h_i, D(stable\text{-}chkpt_i), k\rangle_{\sigma_k} | m' \in in_i\}$,
$m = \langle \text{VIEW-CHANGE}, v, h_i, stable\text{-}chkpt_i, C, P, i\rangle_{\sigma_i}$
$out_i := out_i \cup \{m\}$
$in_i := in_i \cup \{m\}$

RECEIVE$(\langle \text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: let $m = \langle \text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j}$
if $v \geq view_i \wedge correct\text{-}view\text{-}change(m, v, j)$ then
$in_i := in_i \cup \{m\}$

SEND-NEW-VIEW$(v, V)_i$
Pre: $primary(v) = i \wedge v \geq view_i \wedge v > 0 \wedge V \subseteq in_i \wedge |V| = 2f + 1 \wedge \neg has\text{-}new\text{-}view(v, i) \wedge$
$\exists R : (|R| = 2f + 1 \wedge \forall k \in R : (\exists n, s, C, P : (\langle \text{VIEW-CHANGE}, v, n, s, C, P, k\rangle_{\sigma_k} \in V)))$
Eff: $view_i := v$
let $O = \{\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i} | n > max\text{-}n(V) \wedge \exists v' : last\text{-}prepared(m, v', n, merge\text{-}P(V))\}$,
$N = \{\langle \text{PRE-PREPARE}, v, n, null\rangle_{\sigma_i} | max\text{-}n(V) < n < max\text{-}n(O) \wedge$
$\nexists v', m, n : last\text{-}prepared(m, v', n, merge\text{-}P(V))\}$,
$m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_i}$
$seqno_i := max\text{-}n(O)$
$in_i := in_i \cup O \cup N \cup \{m\}$
$out_i := \{m\}$
$update\text{-}state\text{-}nv(i, v, V, m)$
$in_i := in_i - \{\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c} \in in_i | t \leq last\text{-}rep\text{-}t_i(c)\}$

RECEIVE$(\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: let $m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j}$
if $v > 0 \wedge v \geq view_i \wedge correct\text{-}new\text{-}view(m, v) \wedge \neg has\text{-}new\text{-}view(v, i)$ then
$view_i := v$
$out_i := \{\}$
$in_i := in_i \cup O \cup N \cup \{m\}$
for all $\langle \text{PRE-PREPARE}, v, n', m'\rangle_{\sigma_j} \in (O \cup N)$ do
$out_i := out_i \cup \{\langle \text{PREPARE}, v, n', D(m'), i\rangle_{\sigma_i}\}$
if $n' > h_i$ then
$in_i := in_i \cup \{\langle \text{PREPARE}, v, n', D(m'), i\rangle_{\sigma_i}\}$
$update\text{-}state\text{-}nv(i, v, V, m)$
$in_i := in_i - \{\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c} \in in_i | t \leq last\text{-}rep\text{-}t_i(c)\}$

Figure 2-13: View Change Actions

Chapter 3

BFT: An Algorithm Without Signatures

The algorithm in the previous chapter, BFT-PK, is simple but it is slow because it relies on public-key cryptography to sign all messages. Public-key cryptography is the main performance bottleneck in previous Byzantine-fault-tolerant state machine replication systems [Rei94, MR96a, KMMS98]. This chapter describes BFT, a new algorithm that uses message authentication codes (MACs) to authenticate all messages. MACs are based on symmetric cryptography and they can be computed three orders of magnitude faster than signatures. Therefore, the modified algorithm is significantly faster. Additionally, as explained in Chapter 4, the new algorithm eliminates a fundamental problem that prevents BFT-PK from supporting recovery of faulty replicas.

The new algorithm is also interesting from a theoretical perspective because it can be modified to work without relying on cryptography. This can be done by using authenticated point-to-point channels between nodes and by replacing message digests by the message values. With this modification, the algorithm is secure against computationally unbounded adversaries.

The first section in this chapter explains why it is hard to modify BFT-PK to replace signatures by message authentication codes. Section 3.2 presents a description of BFT. An earlier version of this algorithm appeared in [CL99b] and the algorithm in its current form was first presented in [CL00].

3.1 Why it is Hard to Replace Signatures by MACs

Replacing signatures by MACs seems like a trivial optimization but it is not. The problem is that MACs are not as powerful as public-key signatures. For example, in a synchronous system, it is possible to solve the Byzantine consensus problem with any number of faulty participants when using signatures [PSL80]. However, it is necessary to have fewer than one third faulty participants to solve this problem with symmetric authentication [PSL80].

Digital signatures are computed using public-key cryptography. The sender of a message computes a signature, which is a function of the message and the sender's private key, and appends the signature to the message. The receiver can verify the signature using the public key of the sender. Since only the sender knows the signing key and the verification key is public, the receiver can also convince a third party that the message is authentic. It can prove the message was sent by the original sender by simply forwarding the signed message to that third party.

MACs use symmetric cryptography to authenticate the communication between two parties that share a secret session key. The sender of a message computes a MAC, which is a small bit string that is a function of the message and the key it shares with the receiver, and appends the MAC to the message. The receiver can check the authenticity of the message by computing the MAC in the same way and comparing it to the one appended to the message.

MACs are not as powerful as signatures: the receiver may be unable to convince a third party that the message is authentic. This is a fundamental limitation due to the symmetry of MAC computation. The third party is unable to verify the MAC because it does not know the key used to generate it. Revealing the key to the third party does not remove this limitation because a faulty receiver could send messages pretending to be the sender. The other possibility would be for the sender to compute an extra MAC (using a different key shared with the third party) and to append both this MAC and the MAC for the receiver to the message. But this does not work either because a faulty sender could compute a valid MAC for the receiver and an invalid MAC for the third party; since the receiver is unable to check the validity of the second MAC, it could accept the message and not be able to prove its authenticity to the third party.

MACs are sufficient to authenticate messages in many protocols but BFT-PK and previous Byzantine-fault-tolerant algorithms [Rei96, KMMS98] for state machine replication rely on the extra power of digital signatures. BFT-PK is based on the notion of *quorum certificates* and *weak certificates*, which are sets with messages from different replicas. Its correctness relies on the exchange during view changes of certificates collected by the replicas. This works only if the messages in these sets are signed. If messages are authenticated with MACs, a replica can collect a certificate but may be unable to prove to others that it has the certificate.

3.2 The New Algorithm

BFT uses the same system model as BFT-PK and it provides the same service properties. The system model and properties are defined informally in Sections 2.1 and 2.2, and formally in Section 2.4. But BFT uses MACs to authenticate all messages including client requests and replies. Therefore, it can no longer rely on the exchange of prepared, stable and new-view certificates during view changes. We were able to retain the same communication structure during normal case operation and garbage collection at the expense of significant and subtle changes to the view change protocol.

The basic idea behind the new view change protocol is the following: if some non-faulty replica i collects a quorum certificate for some piece of information x, the non-faulty replicas in the quorum can cooperate to send a weak certificate for x to any replica j during view changes. This can be done by having the replicas in the quorum retransmit to j the messages in the certificate they originally sent to i. Since a quorum certificate has at least $2f + 1$ messages and at most f replicas can be faulty, j will eventually receive a weak certificate for the same information x with at least $f + 1$ messages. But weak certificates are not as powerful as quorum certificates. For example, weak prepared certificates can conflict: they can assign the same sequence number to different requests in the same view. The new view change protocol uses invariants that are enforced during normal case operation to decide correctly between conflicting weak certificates.

The use of MACs to authenticate client requests raises additional problems. It is possible for some replicas to be able to authenticate a request while others are unable to do it. This can lead both to safety violations and liveness problems.

Section 3.2.1 explains how messages are authenticated in BFT. Section 3.2.2 describes how the algorithm works when there are no view changes and how it handles authentication of client requests. The new view change protocol is discussed in Section 3.2.4.

3.2.1 Authenticators

The new algorithm uses MACs to authenticate all messages including client requests. 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 for to authenticate communication in both directions. These session keys can be established and refreshed dynamically using the mechanism described in Section 4.3.1 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.

The time to generate and verify signatures is independent of the number of replicas. The time to verify an authenticator is constant but the time to generate one grows linearly with the number of replicas. This is not a problem because we do not expect to have a large number of replicas and there is a large performance gap between MAC and digital signature computation. For example, BFT is expected to perform better than BFT-PK with up to 280 replicas in the experiment described in Section 8.3.3. The size of authenticators also grows linearly with the number of replicas but it grows slowly: it is equal to $8n$ bytes in the current implementation (where n is the number of replicas). For example, an authenticator is smaller than an RSA signature with a 1024-bit modulus for $n \leq 16$ (i.e., systems that can tolerate up to 5 simultaneous faults).

3.2.2 Normal-Case Operation

The behaviors of BFT and BFT-PK are almost identical during normal case operation. The only differences are the following. BFT uses authenticators in request, pre-prepare, prepare, and commit messages and uses a MAC to authenticate replies. The modified protocol continues to ensure the invariant that non-faulty replicas never prepare different requests with the same view and sequence number.

Another difference concerns request authentication. In BFT-PK, backups checked the authenticity of a request when it was about to be executed. Since requests were signed, all replicas would agree either on the client that sent the request or that the request was a forgery. This does not work in BFT because some replicas may be able to authenticate a request while others are unable to do it.

We integrated request authentication into BFT to solve this problem: the primary checks the authenticity of requests it receives from clients and only assigns sequence numbers to authentic requests; and backups accept a pre-prepare message only if they can authenticate the request it contains. A request $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}$ in a pre-prepare message is considered authentic by a backup i in one of the following conditions:

1. the MAC for i in the request's authenticator is correct or
2. i has accepted f prepare messages with the request's digest or
3. i has received a request from client c with the same operation and timestamp and with a correct MAC for i in its authenticator

Condition 1 is usually sufficient for the backups to authenticate requests. But it is possible for the primary to include a request with a corrupt authenticator in a pre-prepare message. This can happen because the client is faulty, the primary is faulty, or the request was corrupted in the network.

A request with an incorrect authenticator may commit provided it has at least $f+1$ correct MACs. Without condition 2, the system could deadlock permanently when this happens. This condition ensures that if a request commits, all backups are eventually able to authenticate it. The condition is safe because the request is not considered authentic unless at least one correct replica was able to verify its MAC in the request's authenticator. It is important for correct replicas to remember requests they pre-prepared across view changes because it may be necessary for them to convince others that requests, which are propagated from previous views, are authentic. Section 3.2.4 explains how this problem is solved.

It is also possible for a request with a corrupt authenticator to force a view change. This may happen when a sequence number is assigned to a request whose authenticator has less than $f + 1$ correct MACs, or when a request is sent to at least one correct backup and the primary is unable to authenticate the request. These view changes are desirable when the cause of the problem is a faulty primary. But they can also be used to mount denial-of-service attacks by replacing correct primaries frequently. Condition 3 allows correct clients to fix the problem by retransmitting the request with a correct authenticator to all the replicas.

However, faulty clients can still force view changes. Our current implementation does not deal with this problem but view changes are sufficiently fast (see Section 8.5) that it is not very serious. We could force suspected clients to sign their requests and replicas could process these requests at lower priority to bound the rate of these view changes.

3.2.3 Garbage Collection

The garbage collection mechanism in BFT is similar to the one in BFT-PK. Replicas collect a stable certificate with checkpoint messages for some sequence number n and then they discard all entries in their log with sequence numbers less than or equal to n and all earlier checkpoints. But since checkpoint messages have authenticators instead of signatures, a weak certificate is insufficient for replicas to prove the correctness of the stable checkpoint during view changes. BFT solves this problem by requiring the stable certificate to be a quorum certificate; this ensures other replicas will be able to obtain a weak certificate proving that the stable checkpoint is correct during view changes.

3.2.4 View Changes

The view change protocol is significantly different in BFT because of the inability to exchange certificates between the replicas. The new protocol is depicted in Figure 3-1. It has the same communication pattern except that backups send acknowledgments to the new primary for each view-change message they receive from another backup. These acknowledgments are used to prove the authenticity of the view-change messages in the new-view certificate.

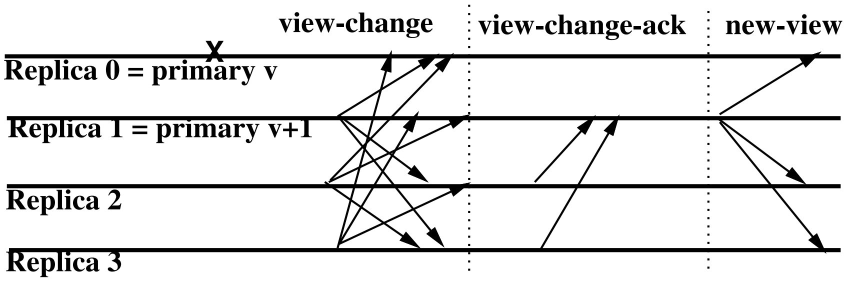

Figure 3-1: View Change Protocol

The basic idea behind the protocol is for non-faulty replicas to cooperate to reconstruct weak certificates corresponding to any prepared or stable certificate that might have been collected by some non-faulty replica in a previous view. This is done by having replicas include in view-change messages information about pre-prepare, prepare, and checkpoint messages that they sent in the past.

We start by describing a simplified view change protocol that may require unbounded space. Section 3.2.5 presents a modification to the protocol that eliminates the problem.

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

The *PSet* at replica i stores information about requests that have prepared at i in previous views. Its entries are tuples $\langle n, d, v\rangle$ meaning that 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.

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

let v be the view before the view change, L be the size of the log, and h be the log's low water mark
for all n such that $h < n \leq h + L$ do
  if request number n with digest d is prepared or committed in view v then
    add $\langle n, d, v\rangle$ to $\mathcal{P}$
  else if $\exists\ \langle n, d', v'\rangle \in$ *PSet* then
    add $\langle n, d', v'\rangle$ to $\mathcal{P}$
  if request number n with digest d is pre-prepared, prepared or committed in view v then
    if $\neg\exists\ \langle n, D\rangle \in$ *QSet* then
      add $\langle n, \{\langle d, v\rangle\}\rangle$ to $\mathcal{Q}$
    else if $\exists\ \langle d, v'\rangle \in D$ then
      add $\langle n,\ D \cup \{\langle d, v\rangle\} - \{\langle d, v'\rangle\}\rangle$ to $\mathcal{Q}$
    else
      add $\langle n, D \cup \{\langle d, v\rangle\}\rangle$ to $\mathcal{Q}$
  else if $\exists\ \langle n, D\rangle \in$ *QSet* then
    add $\langle n, D\rangle$ to $\mathcal{Q}$

Figure 3-2: Computing $\mathcal{P}$ and $\mathcal{Q}$

View-change messages. When a backup i suspects the primary for view v is faulty, it enters view $v + 1$ and multicasts a $\langle \text{VIEW-CHANGE}, v + 1, h, \mathcal{C}, \mathcal{P}, \mathcal{Q}, i\rangle_{\alpha_i}$ 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 sets containing a tuple for every request that is prepared or pre-prepared, respectively, at i. These sets are computed using the information in the log, the *PSet*, and the *QSet*, as explained in Figure 3-2. Once the view-change message has been sent, i stores $\mathcal{P}$ in *PSet*, $\mathcal{Q}$ in *QSet*, and clears its log. The *QSet* may grow without bound if the algorithm changes views repeatedly without making progress. Section 3.2.5 describes a solution to this problem. (It is interesting to note that this problem did not arise in BFT-PK; since prepared certificates contained signed messages, there was no need to maintain information about pre-prepared requests.)

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

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

for all n such that $h < n \leq h + L$ do
  A. if $\exists\, m \in \mathcal{S}$ with $\langle n, d, v\rangle \in m.\mathcal{P}$ that verifies:
    A1. $\exists\, 2f + 1$ messages $m' \in \mathcal{S}$:
      $m'.h < n \wedge \forall\, \langle n, d', v'\rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$
    A2. $\exists\, f + 1$ messages $m' \in \mathcal{S}$:
      $\exists\, \langle n, \{..., \langle d', v'\rangle, ...\}\rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$
    A3. the primary has the request with digest d
  then select the request with digest d for number n

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

Figure 3-3: Decision procedure at the primary.

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 $f + 1$ non-faulty replicas. The last condition is necessary for safety; it ensures that the ordering information for requests that committed with numbers higher than h is still available.

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

The decision procedure ends when the primary has selected a request for each number. After deciding, the primary multicasts a new-view message to the other replicas with its decision. The new-view message has the form $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{X} \rangle_{\alpha_p}$. Here, $\mathcal{V}$ contains a pair for each entry in $\mathcal{S}$ consisting of the identifier of the sending replica and the digest of its view-change message, and $\mathcal{X}$ identifies the checkpoint and request values selected. 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 records all requests in $\mathcal{X}$ as pre-prepared in view $v + 1$ in its log. If it does not have the checkpoint with sequence number h, it also initiates the protocol to fetch the missing state (see Section 5.3.2). In any case the primary does not accept any prepare or commit messages with sequence number less than or equal to h and does not send any pre-prepare message with such a sequence number.

The backups in view $v + 1$ collect messages for view $v + 1$ 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 $\mathcal{S}$ after obtaining a matching view-change certificate, at least $f + 1$ non-faulty replicas can vouch for the authenticity of every view-change message whose digest is in $\mathcal{V}$. Therefore, if the original sender of a view-change is uncooperative the primary retransmits that sender's view-change message and the non-faulty backups retransmit their view-change-acks. A backup can accept a view-change message whose authenticator is incorrect if it receives f view-change-acks that match the digest and identifier in $\mathcal{V}$.

After obtaining the new-view message and the matching view-change messages, the backups check if these messages support the decisions reported by the primary by carrying out the decision procedure in Figure 3-3. If they do not, the replicas move immediately to view $v + 2$. Otherwise, they modify their state to account for the new information in a way similar to the primary. The only difference is that they multicast a prepare message for $v + 1$ for each request they mark as pre-prepared. Thereafter, normal case operation resumes.

The replicas use the status mechanism in Section 5.2 to request retransmission of missing requests as well as missing view-change, view-change acknowledgment, and new-view messages.

Correctness

We now argue informally that the view change protocol preserves safety and that it is live. We will start by sketching a proof of Theorem 3.2.1. This theorem 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.

Theorem 3.2.1 *If a request* m *commits with sequence number* n *at some correct replica in view* v *then the decision procedure in Figure 3-3 will not choose a distinct request for sequence number* n *in any view* $v' > v$

Proof sketch: 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.

In the base case, assume by contradiction that the decision procedure chooses a request $m' \neq m$ for sequence number n in $v' = v + 1$. 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. But since this replica did not garbage collect information for sequence number n, its view-change message must include $\langle n, D(m), v\rangle$ in its $\mathcal{P}$ component. Therefore, condition B cannot be true. Similarly, condition A1 cannot be true for $\langle n, D(m'), v_f\rangle$ because $D(m') \neq D(m)$ (with high probability) and $v_f \leq v$ (because view-change messages for v' are not accepted if there is any tuple with view number greater than $v' - 1$ in their $\mathcal{P}$ component).

The reasoning is similar for the inductive step: $v' > v + 1$. 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 3-2, j's view-change message for v' must include $\langle n, D(m), v_c\rangle$ in its $\mathcal{P}$ component with $v_c \geq v$. 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 $\langle n, \{..., \langle D(m'), v'_c\rangle, ...\}\rangle$ 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 of the theorem.

The primary will also be able to make a correct decision eventually (unless there is a subsequent view change). Assume by contradiction that the primary is unable to do this. Let h_c be the sequence number of the latest checkpoint that is stable at some correct replica. Since this checkpoint is stable, it has been reached by $f + 1$ correct replicas and therefore the primary will be able to choose the value h_c for h. For every sequence number between h and $h + L$, there are two cases: (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 $2f + 1$ non-faulty replicas and non-faulty replicas never prepare different requests for the same view and sequence number; A2 will also be satisfied since a request that prepares at a non-faulty replica pre-prepares at at least $f+1$ non-faulty replicas. Condition A3 may not be satisfied initially, but the primary will eventually receive the request in a response to its status messages (discussed in Section 5.2) and this will trigger the decision procedure to run. Furthermore, since condition A2 is true every replica will be able to authenticate the request that is chosen.

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.

3.2.5 View Changes With Bounded Space

The protocol in the previous section may require an unbounded amount of memory. It bounds the number of tuples in the *QSet* by L but each tuple may grow without bound if there is an unbounded number of view changes before a request with the corresponding sequence number is prepared by a quorum.

This section describes a modified view change protocol that solves this problem. The new protocol bounds the size of each tuple in *QSet*; it retains only pairs corresponding to the M distinct requests that pre-prepared in the latest views where M is a small constant greater than 1 (e.g., 2).

The idea behind the new protocol is the following. When a replica pre-prepares a request with sequence number n in view v, it knows that no distinct request committed in a view earlier than v. But it cannot discard any of the corresponding pairs from the tuple for n in the *QSet* until it can prove this to the other replicas. To obtain these proofs, each replica records this *not-committed* information. Additionally, the protocol delays pre-preparing a request (if that would cause an entry to be discarded from the *QSet*) until the replica obtains messages from a quorum stating that they have matching *not-committed* information. The *not-committed* information is sent in view-change messages; if a replica claims that a request prepared for sequence number n but $f + 1$ replicas say that it did not commit, the new primary can choose a null request for n. The next paragraphs describe the new protocol in more detail.

The new protocol computes the view-change messages as before except that it bounds the size of $\mathcal{Q}$ and the *QSet* as shown in Figure 3-4: if the number of pairs in a tuple exceeds an upper bound, M, the pair with the lowest view number is discarded.

```
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 ≤ h + L do
  if request number n with digest d is prepared or committed in view v then
    add ⟨n, d, v⟩ to P
  else if ∃ ⟨n, d', v'⟩ ∈ PSet then
    add ⟨n, d', v'⟩ to P
  if request number n with digest d is pre-prepared, prepared or committed in view v then
    if ¬∃ ⟨n, D⟩ ∈ QSet then
      add ⟨n, {⟨d, v⟩}⟩ to Q
    else if ∃ ⟨d, v'⟩ ∈ D then
      add ⟨n, D ∪ {⟨d, v⟩} − {⟨d, v'⟩}⟩ to Q
    else
      add ⟨n, D ∪ {⟨d, v⟩}⟩ to Q
      if |D| > M then
        remove entry with lowest view number from D
  else if ∃ ⟨n, D⟩ ∈ QSet then
    add ⟨n, D⟩ to Q
```

Figure 3-4: Computing $\mathcal{P}$ and $\mathcal{Q}$ (with bounded space).

The new protocol has an additional data struture — the *NCset*. Like the others, this set only contains information for sequence numbers between the current low and high water marks in the log. The *NCset* at replica i stores information to prove that certain requests did not commit. Its entries are tuples $\langle n, d, v, u \rangle$ meaning that: d was the digest of request number n proposed in the new-view message with the latest view number v received by i; and no request committed in a view $v' \leq u$ with sequence number n. The view-change messages have an extra field, $\mathcal{NC}$, with the current value of the *NCset*. Replicas only accept a view-change message for view v' provided all tuples $\langle n, d, v, u \rangle$ in its $\mathcal{NC}$ component have $v < v'$ and $u < v$.

Replicas collect view-change and view-change-ack messages as before but the decision procedure used to compute and check the new-view message, which is described in Figure 3-5, is different. It has an extra option, *C*, that enables the new primary to choose a null request for a sequence number if at least one correct replica claims that none of the requests proposed as prepared with that number in $2f + 1$ view-change messages could have committed.

The decision procedure takes $O(L \times |\mathcal{R}|^2 \times M)$ local steps in the worst case and the normal case is much faster because most view-change messages propose identical values, they contain information for less than L requests, and their $\mathcal{Q}$ components contain tuples with less than M pairs.

The *NCSet* is updated when the primary creates a new-view message or a backup accepts a new-view message. This is described in Figure 3-6.

Before sending any prepare message for the requests proposed in a new-view message, each

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

for all n such that $h < n \leq h + L$ do
  A. if $\exists\, m \in \mathcal{S}$ with $\langle n, d, v\rangle \in m.\mathcal{P}$ that verifies:
    A1. $\exists\, 2f+1$ messages $m' \in \mathcal{S}$:
      $m'.h < n \wedge \forall\, \langle n, d', v'\rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$
    A2. $\exists\, f+1$ messages $m' \in \mathcal{S}$:
      $\exists\, \langle n, \{..., \langle d', v'\rangle, ...\}\rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$
    A3. the primary has the request with digest d
  then select the request with digest d for number n

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

  C. else if $2f+1$ messages $m \in \mathcal{S} : m.h < n$
      $\wedge \forall\, \langle n, d, v\rangle \in m.\mathcal{P} : \exists\, f+1$ messages $m' \in \mathcal{S}$:
        $\exists\, \langle n, d', v', u\rangle \in m'.\mathcal{NC} : (d \neq d' \wedge v' > v) \vee u \geq v$
  then select the null request for number n

Figure 3-5: Decision procedure at the primary (with bounded space).

backup i checks if that would cause an entry to be discarded from the *QSet*. In this case, i multicasts a message $\langle$NOT-COMMITTED$, v+1, d, i\rangle_{\alpha_i}$ to all the other replicas where $d = D(\mathcal{V}, \mathcal{X})$ is the digest of the contents of the new-view message. The other replicas reply by multicasting a similar message if they accepted the same new-view message and they have updated their *NCset* according to that message. Backup i waits for not-committed messages from a quorum before sending the prepare messages in $v+1$. This ensures that the not-committed information to justify discarding information from the *QSet* is stored by a quorum and, therefore, will be available in subsequent view changes.

let $\langle$NEW-VIEW$, v+1, \mathcal{V}, \mathcal{X}\rangle_{\alpha_p}$ be the new-view message
for all $\langle n, d\rangle \in \mathcal{X}$ do
  if $\neg\exists\, \langle n, d', v', u\rangle \in$ *NCSet* then
    add $\langle n, d, v+1, 0\rangle$ to *NCSet*
  else if $\exists\, \langle n, d', v', u\rangle \in$ *NCSet* then
    if $d' = d$ then
      NCSet := *NCSet* $- \{\langle n, d', v', u\rangle\} \cup \{\langle n, d, v+1, u\rangle\}$
    else
      NCSet := *NCSet* $- \{\langle n, d', v', u\rangle\} \cup \{\langle n, d, v+1, v'\rangle\}$

Figure 3-6: Computing not-committed information.

Sending a new-view message implicitly pre-prepares a set of requests. Therefore, the new primary p also checks if pre-preparing any of those requests would cause an entry to be discarded from the *QSet*. In this case, p multicasts a message $\langle \text{NOT-COMMITTED-PRIMARY}, v+1, \mathcal{V}, \mathcal{X} \rangle_{\alpha_p}$ to all the backups where $\mathcal{V}$ and $\mathcal{X}$ are the values it intends to send in the new-view message. The backups check the correctness of this message and update their not-committed information as if they were processing a new-view message. Then, they reply by multicasting $\langle \text{NOT-COMMITTED}, v+1, D(\mathcal{V}, \mathcal{X}), i \rangle_{\alpha_i}$ to all other replicas. Once the primary has not-committed messages from a quorum it sends the new-view message.

Processing not-committed messages does not introduce a significant overhead and these messages are sent rarely even for small values of M.

Correctness

The modified view change protocol preserves safety and it is live. We will first argue that Theorem 3.2.1 is true by reusing the proof that was presented at the end of the previous section. For the modified protocol to make the theorem false, condition C must be true for a sequence number n in view v' after a request commits with sequence number n in a view $v < v'$. The proof is by induction on the number of views between v and v'. In the base case ($v' = v+1$), condition C cannot be true because replicas do not accept view-change messages for view $v+1$ unless all tuples $\langle n, d, v'', u \rangle$ in their $\mathcal{NC}$ component have $u < v'' \leq v$. For the inductive step, condition C cannot be true because the inductive hypothesis and the procedure to update the *NCSet* imply that no correct replica can send a view-change message with $\langle n, d', v'', u \rangle$ with $u \geq v$ or $d' \neq D(m) \wedge v'' > v$.

The modified protocol also enables the primary to eventually make the correct decision. Discarding information from the *QSet* could potentially prevent progress: a correct replica could prepare a request with sequence number n and another correct replica could discard information that the request had pre-prepared. This could prevent the primary from making a decision because neither condition A2 nor condition B would ever be true. The new protocol prevents the problem because when a correct replica drops information for sequence number n from its *QSet* there is not-committed information justifying its action in the *NCSet* of all correct replicas in a quorum. Therefore, condition C will be true for sequence number n if neither condition A nor B can be true.

Chapter 4

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. We 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 window of vulnerability.

Limiting the number of faults that can occur in a finite window is a synchrony assumption but such an assumption is unavoidable: since Byzantine-faulty replicas can discard the service state, it is necessary to bound the number of failures that can occur before recovery completes. To tolerate f faults over the lifetime of the system, BFT-PR requires no synchrony assumptions.

By making recoveries automatic, the window of vulnerability can be made very small (e.g., a few minutes) with low impact on performance. Additionally, our algorithm provides *detection* of denial-of-service attacks aimed at increasing the window; replicas can time how long a recovery takes and alert their administrator if it exceeds some pre-established bound. The administrator can then take steps to allow recovery to complete. Therefore, integrity can be preserved even when there is a denial-of-service attack. Furthermore, the algorithm detects when the state of a replica is corrupted by an attacker and can log the differences between the corrupt state and the state of non-faulty replicas. This information can be valuable to analyze the attack and patch the vulnerability it exploited.

Section 4.1 presents an overview of the problems that arise when providing recovery from Byzantine faults. Section 4.2 describes the additional assumptions required to provide automatic recoveries and the modifications to the algorithm are described in Section 4.3.

4.1 Overview

The recovery mechanism embodies several new techniques needed to solve the problems that arise when providing recovery from Byzantine faults:

Proactive recovery. A Byzantine-faulty replica may appear to behave properly even when broken;

therefore recovery must be proactive to prevent an attacker from compromising the service by corrupting $1/3$ of the replicas without being detected. Our algorithm recovers replicas periodically independent of any failure detection mechanism. However, a recovering replica may not be faulty and recovery must not cause it to become faulty, since otherwise the number of faulty replicas could exceed the bound required to provide correctness. In fact, we need to allow the replica to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery.

Fresh messages. An attacker must be prevented from impersonating a replica that was faulty after it recovers. Impersonation can happen if the attacker learns the keys used to authenticate messages. But even if messages are signed using a secure cryptographic co-processor, an attacker will be able to sign bad messages while it controls a faulty replica. These bad messages could be replayed later to compromise safety. To solve this problem, we define a notion of authentication *freshness* and replicas reject messages that are not fresh. As a consequence, replicas may be unable to prove to a third party that some message they received is authentic because it may no longer be fresh. BFT can support recovery because it does not rely on such proofs but BFT-PK and all previous state-machine replication algorithms [Rei95, KMMS98] relied on them.

Efficient state transfer. State transfer is harder in the presence of Byzantine faults and efficiency is crucial to enable frequent recovery with low degradation of service performance. To bring a recovering replica up to date, the state transfer mechanism must check the local copy of the state to determine which portions are both up-to-date and not corrupt. Then, it must ensure that any missing state it obtains from other replicas is correct. We have developed an efficient hierarchical state transfer mechanism based on Merkle trees [Mer87] and incremental cryptography [BM97]; the mechanism tolerates Byzantine-faults and modifications to the state while transfers are in progress. It is described in Section 5.3.2.

4.2 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 implement automatic recoveries we need additional assumptions:

Secure Cryptography. Each replica has a secure cryptographic co-processor, e.g., a Dallas Semiconductors iButton or the security chip in the motherboard of the IBM PC 300PL. The co-processor stores the replica's private key, and can sign and decrypt messages without exposing this key. It also contains a true random number generator, e.g., based on thermal noise, and a counter that never goes backwards. This enables it to append random numbers or the counter to messages it signs.

Read-Only Memory. Each replica stores the public keys for other replicas in some memory that survives failures without being corrupted (provided the attacker does not have physical access to the machine). This memory could be a portion of the flash BIOS. Most motherboards can be configured such that it is necessary to have physical access to the machine to modify the BIOS.
Watchdog Timer. Each replica has a *watchdog timer* that periodically interrupts processing and hands control to a *recovery monitor*, which is stored in the read-only memory. For this mechanism to be effective, an attacker should be unable to change the rate of watchdog interrupts without physical access to the machine. Some motherboards and extension cards offer the watchdog timer functionality but allow the timer to be reset without physical access to the machine. However, this is easy to fix by preventing write access to control registers unless some jumper switch is closed.

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

Note that all previous proactive security algorithms [OY91, HJKY95, HJJ$^+$97, CHH97, GGJR99] 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 co-processors to establish new session keys between the replicas after a recovery.

The only work on proactive security that does not assume authenticated channels is [CHH97], 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 co-processors with the properties we require are now readily available, e.g., IBM is selling PCs with a cryptographic co-processor in the motherboard at essentially no added cost. We also assume clients have a secure co-processor; this simplifies the key exchange protocol between clients and replicas but it could be avoided by adding an extra round to this protocol.

4.3 Modified Algorithm

Recall that in BFT replicas collect certificates. Correctness requires that certificates contain at most f messages that were sent by replicas when they were faulty. Recovery complicates the collection of certificates. If a replica collects messages for a certificate over a sufficiently long period of time, it can end up with more than f messages from faulty replicas. We avoid this problem by changing keys periodically and by having replicas reject messages that are authenticated with old keys. This is explained in Section 4.3.1 and the recovery mechanism is discussed in Section 4.3.2.

4.3.1 Key Exchanges

Replicas and clients refresh the session keys used to send messages to them by sending *new-key* messages periodically (e.g., every minute). The same mechanism is used to establish the initial session keys. The message has the form $\langle \text{NEW-KEY}, i, ..., \{k_{j,i}\}_{\epsilon_j}, ..., t\rangle_{\sigma_i}$. The message is signed by the secure co-processor (using the replica's private key) and t is the value of its counter; the counter is incremented by the co-processor and appended to the message every time it generates a signature. (This prevents suppress-replay attacks [Gon92].) Each $k_{j,i}$ is the key replica j should use to authenticate messages it sends to i in the future; $k_{j,i}$ is encrypted by j's public key, so that only j can read it. Replicas use timestamp t to detect spurious new-key messages: t must be larger than the timestamp of the last new-key message received from i.

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

When a replica or client sends a new-key message, it discards all messages in its log that are not part of a complete certificate (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, i.e., messages authenticated with keys created in the same refreshment epoch.

4.3.2 Recovery

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

Reboot. Recovery is proactive — it starts periodically when the watchdog timer goes off. 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.

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 saving its state and rebooting; 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.

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+1st fault. But if the recovering replica was faulty, the state may be corrupt and the attacker may forge messages because it knows the MAC keys used to authenticate both incoming and outgoing messages. The rest of the recovery protocol solves these problems.

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

Run estimation protocol. Next, i runs a simple protocol to estimate an upper bound, H_M, on the high-water mark that it would have in its log if it were not faulty; it discards any log entries or checkpoints with greater sequence numbers to bound the sequence number of corrupt information in its state. 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 received 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, query-stable, and status messages (see Section 5.2).

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 insured because if a checkpoint is stable, it will have been created by at least $f + 1$ non-faulty replicas and it will have a sequence number less than or equal to any value of c that they propose. The test against p ensures that c_M is close to a checkpoint at some non-faulty replica since at least one non-faulty replica reports a p not less than c_M; this is important because it prevents a faulty replica from prolonging i's recovery. Estimation is live because there are $2f + 1$ non-faulty replicas and they only propose a value of c if the corresponding request committed; 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 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 co-processor and t is the co-processor's counter to prevent replays. The other replicas reject the request if it is a replay or if they accepted a recovery request from i recently (where recently can be defined as half of the watchdog period). This is important to prevent a denial-of-service attack where non-faulty replicas are kept busy executing recovery requests.

The recovery request is treated like any other request: it is assigned a sequence number n_R and it goes through the usual three phases. But when another replica executes the recovery request, it sends its own new-key message. Replicas also send a new-key message when they fetch missing state (see Section 5.3.2) 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, i.e., $H_R = \lfloor n_R/K \rfloor \times K + L$.

The reply to the recovery request includes the sequence number n_R. Replica i uses the same protocol as the client to collect the correct reply to its recovery request but waits for $2f + 1$ replies. Then it computes its *recovery point*, $H = max(H_M, H_R)$. 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.

The mechanism to compute a valid view ensures that non-faulty 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. Since the recovery point is greater than or equal to H_R, it will be greater than the sequence number of any request that propagated to v_r from an earlier view.

Check and fetch state. While i is recovering, it uses the state transfer mechanism discussed in Section 5.3.3 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 the checkpoint with sequence number H is stable. This ensures that any state other replicas relied on i to have is actually held by $f + 1$ non-faulty replicas. Therefore if some other replica fails now, we can be sure the state of the system will not be lost. This is true because the estimation procedure run at the beginning of recovery ensures that while recovering i never sends bad messages for sequence numbers above the recovery point. Furthermore, the recovery request ensures that other replicas will not accept forged messages with sequence numbers greater than H.

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. This allows replicas to estimate the duration of i's recovery, which is useful to detect denial-of-service attacks that slow down recovery with low false positives.

4.3.3 Improved Service Properties

Our system ensures safety and liveness (as defined in Section 2.2) for an execution τ provided at most f replicas become faulty within a window of vulnerability of size $T_v = 2T_k + T_r$. The values of T_k and T_r are characteristic of each execution τ and unknown to the algorithm. T_k is the maximum key refreshment period in τ for a non-faulty node, and T_r is the maximum time between when a replica fails and when it recovers from that fault in τ.

The session key refreshment mechanism from Section 4.3.1 ensures non-faulty nodes only accept certificates with messages generated within an interval of size at most $2T_k$.[1] The bound on the number of faults within T_v ensures there are never more than f faulty replicas within any interval of size at most $2T_k$. Therefore, safety and liveness are provided because non-faulty nodes never accept certificates with more than f bad messages.

Because replicas discard messages in incomplete certificates when they change keys, BFT-PR requires a stronger synchrony assumption in order to provide liveness. It assumes there is some unknown point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time d or all non-faulty clients have received replies to their requests; here, d is a constant that depends on the timeout values used by the algorithm to refresh keys, and trigger view-changes and recoveries.

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

We have little control over the value of T_v because T_r may be increased by a denial-of-service attack. But we have good control over T_k and the maximum time between watchdog timeouts, T_w, because their values are determined by timer rates, which are quite stable. Setting these timeout values involves a tradeoff between security and performance: small values improve security by reducing the window of vulnerability but degrade performance by causing more frequent recoveries and key changes. Section 8.6.3 analyzes this tradeoff and shows that these timeouts can be quite small with low performance degradation.

The period between key changes, T_k, can be small without impacting performance significantly (e.g., 15 seconds). But T_k should be substantially larger than 3 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 non-faulty replica under normal load conditions. There is no point in recovering a replica when its previous recovery has not yet finished; and we stagger the recoveries so that no more than f replicas are recovering at once, since otherwise service could be interrupted even without an attack. Therefore, we set $T_w = 4 \times s \times R_n$. Here, the factor 4 accounts for the staggered recovery of $3f + 1$ replicas f at a time, and s is a safety factor to account for benign overload conditions (i.e., no attack).

The results in Section 8.6.3 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.

Chapter 5

Implementation Techniques

We developed several important techniques to implement BFT efficiently. This chapter describes these techniques. They range from protocol optimizations to protocol extensions that enable replication of some non-deterministic services. The protocol optimizations are described in Section 5.1. Section 5.2 explains a message retransmission mechanism that is well-suited for BFT and Section 5.3 explains how to manage checkpoints efficiently. The last two sections describe how to handle non-deterministic services and how to defend against denial of service attacks.

5.1 Optimizations

This section describes several optimizations that improve the performance during normal case operation while preserving the safety and liveness properties. The optimizations can all be combined and they can be applied to BFT-PK as well as BFT (with or without recovery).

5.1.1 Digest Replies

The first 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. This optimization is not used for very small replies; the threshold in the current implementation is set to 32 bytes.

This optimization is very effective when combined with request batching (see Section 5.1.4). It enables several clients to receive large replies in parallel from different replicas. As a result, the aggregate throughput from the service to the clients can be several times above the maximum link bandwidth. The optimization is also important at reducing protocol overhead when the number of replicas increases: it makes the overhead due to additional replicas independent of the size of the operation result.

5.1.2 Tentative Execution

The second optimization reduces the number of message delays for an operation invocation from 5 to 4. Replicas execute requests *tentatively*. A request is executed as soon as the following conditions are satisfied: the replicas have a prepared certificate for the request; their state reflects the execution of all requests with lower sequence number; and these requests are all known to have committed. After executing the request, the replicas send tentative replies to the client.

Since replies are tentative, the client must wait for a quorum certificate with replies with the same result before it accepts that result. This ensures that the request is prepared by a quorum and, therefore, it is guaranteed to commit eventually at non-faulty replicas. If the client's retransmission timer expires before it receives these replies, the client retransmits the request and waits for a weak certificate with non-tentative replies. Figure 5-1 presents an example tentative execution.

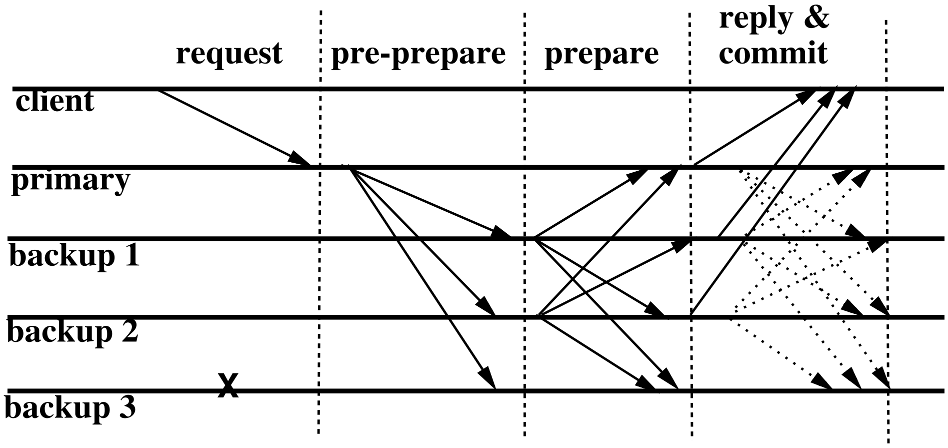

Figure 5-1: Tentative execution

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

Replicas checkpoint their state immediately after executing a request, whose sequence number is divisible by the checkpoint interval, tentatively. But they only send a checkpoint message after the request commits.

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 both load on the network and on the replicas' CPUs. However, it has a low impact on the latency of the service because, with tentative execution, the commit phase is already overlapped with the sending of new requests to the service. Its impact on throughput is also low because the batching optimization described in Section 5.1.4 amortizes the cost of the commit phase over many requests.

5.1.3 Read-only Operations

The next 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, that the client has access, and that the request is in fact read-only. The last two checks are performed by a service specific upcall. The last check is important because a faulty client could mark as read-only a request that modifies the service state.

A replica sends back a reply only after all requests reflected in the state in which it executed the read-only request have committed; this is necessary to prevent the client from observing uncommitted state that may be rolled back. 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. This optimization reduces latency to a single round-trip for read-only requests as depicted in Figure 5-2.

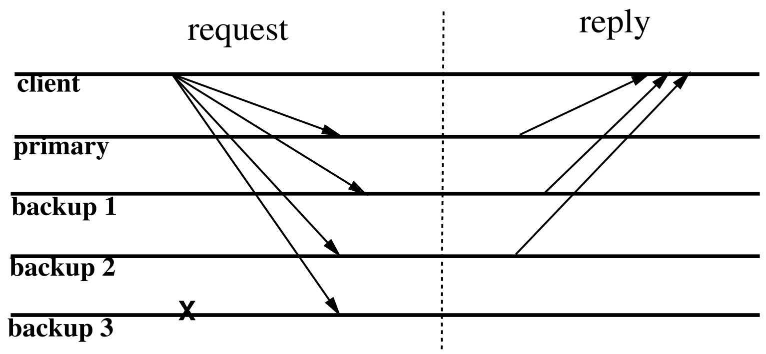

Figure 5-2: Read-only operations

The read-only optimization preserves the modified linearizability condition. To show this, we will argue that any read-only operation o can be serialized after any operation that ends before o starts and before any operation that starts after o ends. (An operation starts when the request to execute it is sent for the first time and ends when the client obtains the result.)

Let Q be the quorum certificate containing the replicas that send the replies with o's result. When any read-write operation, p, that precedes o ends, it has been tentatively executed by a quorum Q'. Therefore, any write performed by p will be reflected in o's result because Q' intersects Q in at least one correct replica. Similarly, any operation that starts after o ends will return a result that reflects all the writes observed by o and maybe later writes. This is true because o's results do not reflect uncommitted state and Q' intersects in at least one correct replica the quorum that tentatively executes any later read-write operation or the quorum that sends replies to any later read-only operation.

Note that for the read-only optimization to work correctly, it is required that the client obtain a quorum certificate with replies not only for read-only operations but also for any read-write operation. This is the case when replies are tentative but the algorithm must be modified for this to happen with non-tentative replies (before it was sufficient to obtain a weak certificate). This is generally a good tradeoff; the only exception are environments with a high message loss rate.

5.1.4 Request Batching

The algorithm can process many requests in parallel. The primary can send a pre-prepare with a sequence number assignment for a request as soon as it receives the request; it does not need to wait for previous requests to execute. This is important for networks with a large bandwidth-delay product but, when the service is overloaded, it is better to process requests in batches.

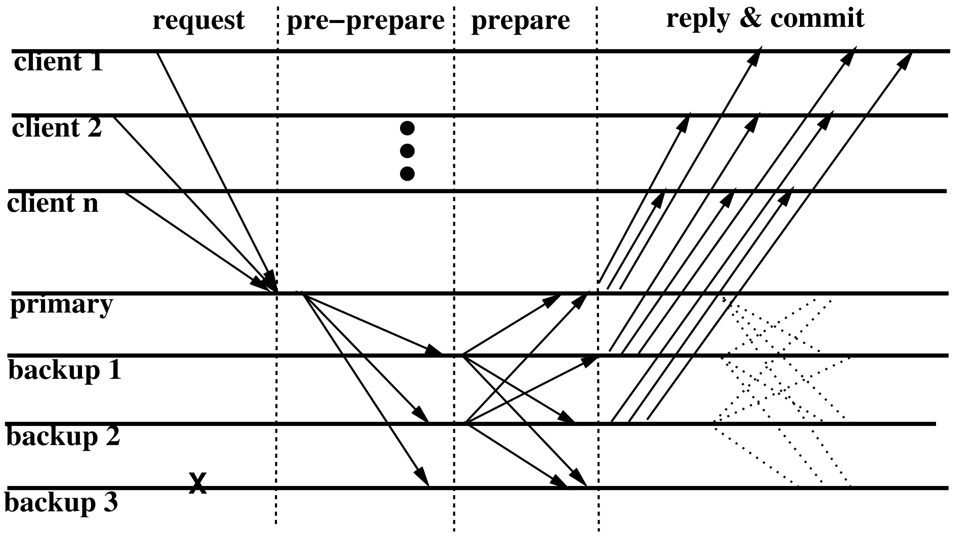

Figure 5-3: Request batching

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 normal case protocol for the batch; this optimization is similar to a group commit in transactional systems [GK85]. Figure 5-3 depicts the processing of a batch of requests.

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. 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.

Our batching mechanism reduces both CPU and network overhead under load without increasing the latency to process requests in an unloaded system. Previous state machine replication systems that tolerate Byzantine faults [MR96a, KMMS98] have used batching techniques that impact latency significantly.

5.1.5 Separate Request Transmission

The algorithm we described inlines requests in pre-prepare messages. This simplifies request handling but it leads to higher latency for large requests because they go over the network twice: the client sends the request to the primary and then the primary sends the request to the backups in a pre-prepare message. Additionally, it does not allow request authentication and digest computation to be performed in parallel by the primary and the backups: the primary authenticates requests before it sends the pre-prepare message and the backups authenticate requests when they receive this message.

We modified the algorithm not to inline requests whose size is greater than a threshold (currently 255 bytes), in pre-prepare messages. Instead, the clients multicast these requests to all replicas; replicas authenticate the requests in parallel; and they buffer those that are authentic. The primary selects a batch of requests to include in a pre-prepare message (as described in the previous section) but it only includes their digests in the message. This reduces latency for operations with large arguments and it also improves throughput because it increases the number of large requests that can be batched in a single pre-prepare message.

5.2 Message Retransmission

BFT is implemented using low-level, unreliable communication protocols, which may duplicate or lose messages or deliver them out of order. The algorithm tolerates out-of-order delivery and rejects duplicates. This Section describes a technique to recover from lost messages.

It is legitimate to ask why BFT does not use an existing reliable communication protocol. There are many protocols in the literature to implement reliable point-to-point (e.g., TCP [Pos81]) and multicast communication channels (e.g., XTP [SDW92]). These protocols ensure that messages sent between correct processes are eventually delivered but they are ill-suited for algorithms that tolerate faults in asynchronous systems. The problem is that any reliable channel implementation requires messages to be buffered until they are known to have been received. Since a faulty receiver cannot be distinguished from a slow one in an asynchronous system, any reliable channel implementation requires either an unbounded amount of buffer space or requires the algorithm to stop when buffer space runs out due to a faulty receiver.

BFT uses a receiver-based mechanism inspired by the SRM [FJL$^+$95] framework to recover from lost messages in the communication between replicas: a replica i multicasts small *status* messages that summarize its state; when other replicas receive a status message they retransmit messages they have sent in the past that i is missing using unicast. Status messages are sent periodically and when the replica detects that it is missing information (i.e., they also function as negative acknowledgments).

This receiver-based mechanism works better than a sender-based one because it eliminates unnecessary retransmissions. The sender can use the summary of the receiver's state to avoid retransmitting messages that are no longer required for the receiver to make progress. For example, assume replica j sent a prepare message p to i, which was lost, but i prepared the request corresponding to p using messages received from other replicas. In this case, i's status message will indicate that the request is prepared and j will not retransmit p. Additionally, this mechanism eliminates retransmissions to faulty replicas.

The next paragraphs describe the mechanism BFT uses to recover from lost messages in more detail. A replica i whose current view v is active multicasts messages with the format $\langle \text{STATUS-ACTIVE}, v, h, le, i, P, C\rangle_{\alpha_i}$. Here, h is the sequence number of the last stable checkpoint, le is the sequence number of the last request i has executed, P contains a bit for every sequence number between le and H (the high water mark in the log) indicating whether that request prepared at i, and C is similar but indicates whether the request committed at i.

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

If a replica j is unable to validate the status message, it sends its last new-key message to i. Otherwise, j sends messages it sent in the past that i may require in order to make progress. For example, if i is in a view less than j's, j sends i its latest view-change message. In all cases, j authenticates messages it retransmits with the latest keys it received in a new-key message from i. This is important to ensure liveness with frequent key changes.

BFT uses a different mechanism to handle communication between clients and replicas. The receiver-based mechanism does not scale well to a large number of clients because the information about the last requests received from each client grows linearly with the number of clients. Instead, BFT uses an adaptive retransmission scheme [KP91] similar to the one used in TCP. Clients retransmit requests to replicas until they receive enough replies. They measure response times to compute the retransmission timeout and use a randomized exponential back off if they fail to receive a reply within the computed timeout. If a replica receives a request that has already been executed, it retransmits the corresponding reply to the client.

5.3 Checkpoint Management

BFT's garbage collection mechanism (see Section 2.3.4) 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 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.

5.3.1 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 non-leaf partition is divided into s equal-sized, contiguous sub-partitions. Figure 5-4 depicts a partition tree with three levels. We call the leaf partitions *pages* and the interior ones meta-data. For example, the experiments described in Chapter 8 were run with a hierarchy with four levels, s equal to 256, and 4KB pages.

Each replica maintains one logical copy of the partition tree for each checkpoint. The copy is created when the checkpoint is taken and it is discarded when a later checkpoint becomes stable. 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 meta-data partition and a tuple $\langle lm, d, p\rangle$ for each page. Here, lm is the sequence number of the checkpoint at the end of the last checkpoint 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

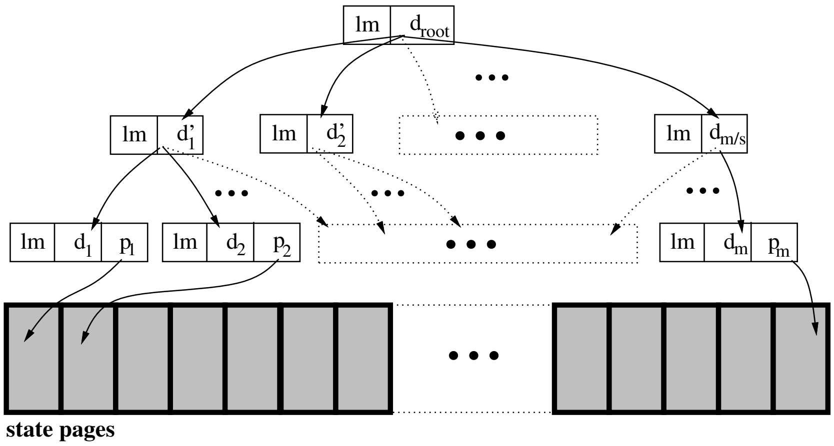

Figure 5-4: Partition tree.

cryptographic hash function (currently MD5 [Riv92]) to the string obtained by concatenating the index of the page within the state, its value of lm, and p. A meta-data 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 sub-partitions. Thus, we apply AdHash [BM97] at each meta-data level. This construction has the advantage that the digests for a checkpoint can be obtained efficiently by updating the digests from the previous checkpoint incrementally. It is inspired by Merkle trees [Mer87].

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.

5.3.2 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 non-leaf 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. The idea is that the digest of a partition commits the values of all its sub-partitions. 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 sub-partitions it fetches. The replica does not need a weak certificate for the sub-partitions unless the value of a sub-partition 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 -1 or it specifies that i is seeking the value of the partition at sequence number c from replica k.

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

If $\langle \text{FETCH}, l, x, lc, c, k, i \rangle_{\alpha_i}$ is received by the designated replier, k, and it has a checkpoint for sequence number c, it sends back $\langle \text{META-DATA}, c, l, x, P, k \rangle$, where P is a set with a tuple $\langle x', lm, d \rangle$ for each sub-partition of (l, x) with index x', digest d, and $lm > lc$. Since i knows the correct digest for the partition value at checkpoint c, it can verify the correctness of the reply without the need for 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 sub-partition values for the same sequence number cp (greater than lc and c). Then, it compares its digests for each sub-partition of (l, x) with those in the fetched information; it multicasts a fetch message for sub-partitions where there is a difference, and sets the value in $\mathcal{LC}$ to c (or cp) for the sub-partitions that are up to date. Since i learns the correct digest of each sub-partition at checkpoint c (or cp), it can use the optimized protocol to fetch them using the digests to ensure their correctness.

The protocol recurses down the tree until i sends fetch messages for out-of-date pages. Pages are fetched like other partitions except that meta-data replies contain the digest and last modification sequence number for the page rather than sub-partitions, and the designated replier sends back

$\langle \text{DATA}, x, p \rangle$. Here, x is the page index and p is the page value. The protocol imposes little overhead on other replicas; only one replica replies with the full page and it does not even need to compute a MAC for the message since i can verify the reply using the digest it already knows.

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

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

Since state transfer happens concurrently with request execution at other replicas and other replicas are free to garbage collect checkpoints, it may take some time for a replica to complete the protocol, e.g., each time it fetches a missing partition, it receives information about yet a later modification. If the service operations change data faster than it can be transfered, 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 ever is actually needed (because others have failed), the system will wait for it to catch up.

5.3.3 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 meta-data partition is set to -1. This ensures that the meta-data replies include digests for all sub-partitions.

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 was 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.

5.4 Non-Determinism

State machine replicas must be deterministic but many services involve some form of non-determinism. For example, the time-last-modified in a distributed file system is set by reading the server's local clock; if this were done independently at each replica, the states of non-faulty replicas would diverge. This section explains how to extend the algorithm to allow replication of such services.

The idea is to modify the service code to remove the computations that make non-deterministic choices. Replicas run a protocol to agree on the value of these choices for each operation and this value is passed as an argument to the operation. In general, the client cannot select the value because it does not have enough information; for example, it does not know how its request will be ordered relative to concurrent requests by other clients. Instead the primary selects the value independently or based on values provided by the backups.

If the primary selects the non-deterministic value independently, it concatenates the value with the associated request batch and sends the value and the batch in a pre-prepare message. Then, it runs the three phase protocol to ensure that non-faulty replicas agree on a sequence number for the request batch and the value. This prevents a faulty primary from causing replica state to diverge by sending different values to different backups. However, a faulty primary might send the same, incorrect, value to all backups. Therefore, when the backups are about to execute the request, they check the value proposed by the primary. If this value is correct, they execute the request; otherwise, they can choose an alternative or reject the request. But they must be able to decide deterministically whether the value is correct (and what to do if it is not); their decision must be completely determined by the service state and operation arguments.

This protocol is adequate for most services (including the NFS service in Section 6.3) but occasionally backups must participate in selecting the values to satisfy a service's specification, e.g., in services that generate a timestamp that must be close to real time. This can be accomplished by adding an extra phase to the protocol: the primary obtains authenticated values proposed by the backups, concatenates $2f+1$ of them with the associated request batch, and starts the three phase protocol for the concatenated message. Replicas choose the value by a deterministic computation on the $2f+1$ values and their state, e.g., taking the median ensures that the chosen value is between the values proposed by two non-faulty replicas.

It may be possible to optimize away the extra phase in the common case. For example, if replicas need a time value that is "close enough" to that of their local clock, the extra phase can be avoided when their clocks are synchronized within some delta. Replicas can check the value proposed by the primary in the pre-prepare message and reject this message if the value is not close to their local clock. A primary that proposes bad values is replaced as usual by the view change mechanism.

5.5 Defenses Against Denial-Of-Service Attacks

The most important defense against denial-of-service attacks is to avoid making synchrony assumptions. BFT does not rely on any synchrony assumption to provide safety. Therefore, a denial-of-service attack cannot cause a replicated service to return incorrect replies. But it can prevent the service from returning replies by exhausting resources at the replicas or the network.

We implemented several defenses to make denial-of-service attacks harder and to ensure that systems can continue to provide correct service after an attack ends. The idea is to manage resources carefully to prevent individual clients or replicas from monopolizing any resource. The defenses include using inexpensive message authentication, bounding the rate of execution of expensive operations, bounding the amount of memory used, and scheduling client requests fairly.

Replicas only accept messages that are authenticated by a known client or another replica; other messages are immediately rejected. This can be done efficiently because most message types use MACs that are inexpensive to compute. The only exception are new-key messages and recovery requests, which are signed using public-key cryptography. Since correct replicas and clients only send these messages periodically, replicas can discard these messages without even checking their signatures if the last message from the same principal was processed less than a threshold time before. This bounds the rate of signature verification and the rate at which authentic messages from faulty principals are processed, which is important because they they are expensive to process.

The amount of memory used by the algorithm is bounded: it retains information only about sequence numbers between the low and high water mark in the log, and it bounds the amount of information per sequence number. Additionally, it bounds the fraction of memory used on behalf of any single client or replica. For example, it retains information about a single pre-prepare, prepare, or commit message from any replica for the same view and sequence number. This ensures that the algorithm always has enough memory space to provide service after an attack ends.

To ensure that client requests are scheduled fairly, the algorithm maintains a FIFO queue for requests waiting to be processed and it retains in the queue only the request with the highest timestamp from each client. If the current primary does not schedule requests fairly, the backups trigger a view change. The algorithm defends against attacks that replay authentic requests by caching the last reply sent to each client and the timestamp, t, of the corresponding request. Requests with timestamp lower than t are immediately discarded and replicas use the cached reply to handle requests with timestamp t efficiently.

Chapter 6

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 6.1 describes the library's implementation and Section 6.2 presents its interface. We used the library to implement a Byzantine-fault-tolerant NFS file system, which is described in Section 6.3.

6.1 Implementation

The library uses a connectionless model of communication: point-to-point communication between nodes is implemented using UDP [Pos80], and multicast to the group of replicas is implemented using UDP over IP multicast [DC90]. 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 Section 2.4. 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.

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 pre-conditions become true. The handlers never block waiting for messages.

We use the SFS [MKKW99] implementation of a Rabin-Williams public-key cryptosystem with a 1024-bit modulus to establish 128-bit session keys. All messages are then authenticated using message authentication codes computed using these keys and UMAC32 [BHK$^+$99]. Message digests are computed using MD5 [Riv92].

The implementation of public-key cryptography signs and encrypts messages as described in [BR96] and [BR95], respectively. These techniques are provably secure in the *random oracle model* [BR95]. In particular, signatures are non-existentially 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 a more secure hash function (for example, SHA-1 [SHA94]) at the expense of some performance degradation.

We have described our protocol messages at a logical level without specifying the size or layout of the different fields. We believe that it is premature to specify the detailed format of protocol messages without further experimentation. But to understand the performance results in the next two chapters, it is important to describe the format of request, reply, pre-prepare, and prepare messages in detail. Figure 6-1 shows these formats in our current implementation.

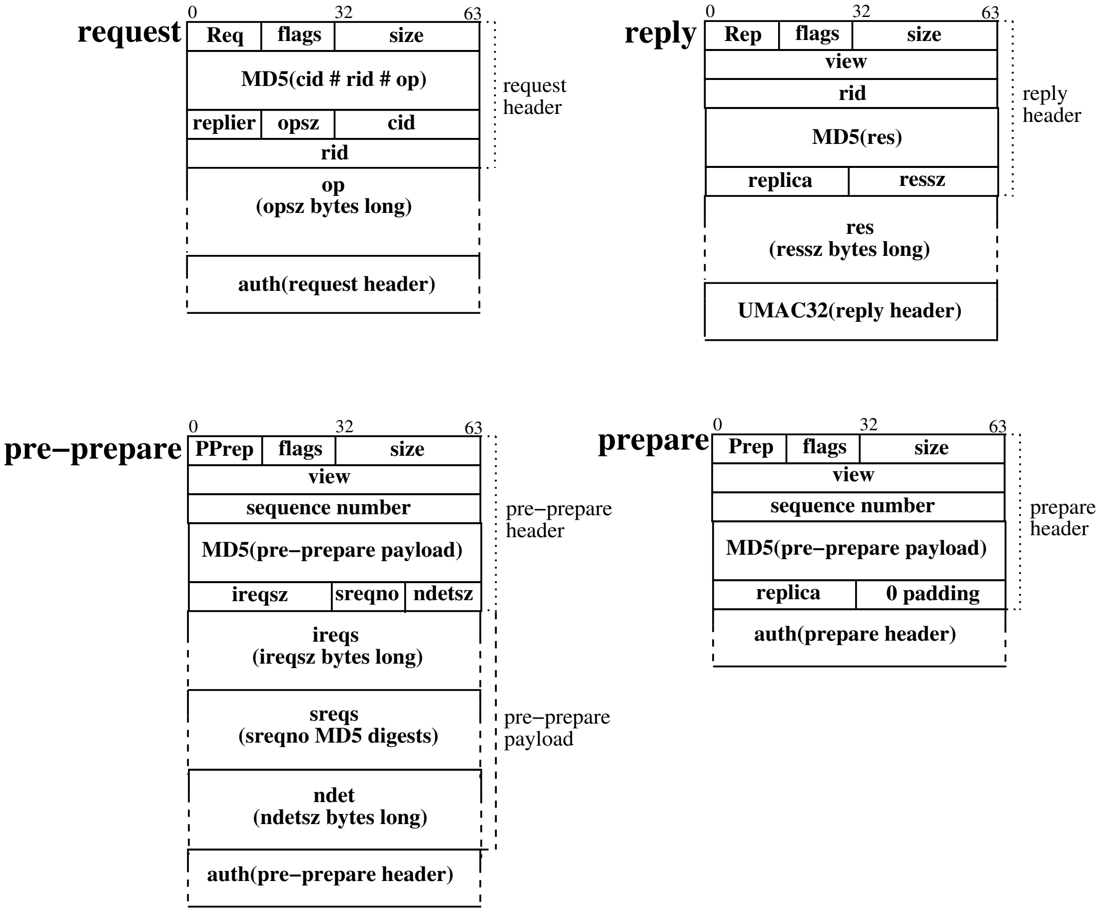

Figure 6-1: Message formats.

All protocol messages have a generic 64-bit header, which contains a tag that identifies the message type, a set of flags that are type specific, and the total size of the message. The generic header is part of a type-specific header, which has a fixed size for each type.

The request header includes an MD5 digest of the string obtained by concatenating the client identifier, the request identifier (timestamp), and the operation being requested. The header also includes the identifier of the designated replier (that is the replica chosen to return the result in the digest-replies optimization), the size of the operation in bytes, *opsz*, the client identifier, *cid*, and the request identifier, *rid*. The flags in the request header indicate whether to use the read-only optimization and whether the request contains a signature or an authenticator. In the normal case, all requests contain authenticators. In addition to the header, the request message includes a variable size payload with the operation being requested and an authenticator. The authenticator is composed of a 64-bit nonce, and n 64-bit UMAC32 tags that authenticate the request header (where n is the number of replicas). When a replica receives a request, it checks if the corresponding MAC in the authenticator and the digest in the header are correct.

The primary assigns a sequence number to a batch of requests and sends a pre-prepare message. The pre-prepare header includes the primary's view number, the sequence number, an MD5 digest of the pre-prepare payload, the number of bytes in requests inlined in the message, *ireqsz*, the number of digests of requests that are not inlined, *sreqno*, and the number of bytes in the non-deterministic value associated with the batch, *ndetsz*. The variable size payload includes the requests that are inlined, *ireqs*, the digests in the headers of the remaining requests in the batch, *sreqs*, and the non-deterministic choices, *ndet*. Additionally, the message includes an authenticator with a nonce, and $n - 1$ UMAC32 tags that authenticate the pre-prepare header.

The current implementation limits the total size of pre-prepare messages to 9000 bytes (to fit in a UDP message in most kernel configurations) and the number of request digests to 16 (to limit the amount of storage used up by the log). This limits the batch size.

When the backups receive a pre-prepare message they check if the corresponding MAC in the authenticator and the digest in the header are correct. They also check the requests that are inlined in the message. The requests that are transmitted separately are usually checked in parallel by the primary and the backups.

If the backups accept the pre-prepare message and they have already accepted the requests in the batch that are transmitted separately, they send a prepare message. The prepare header includes the view number, the sequence number, an MD5 digest of the pre-prepare payload, the identifier of the backup, and it is padded with 0's to a 64-bit boundary. The message has an authenticator with a nonce, and $n - 1$ UMAC32 tags that authenticate the prepare header. When the replicas receive a prepare message, they check the corresponding MAC in the authenticator.

Once the replicas have the pre-prepare and at least $2f$ prepare messages with the same digest in the header, they execute all operations in the batch tentatively and send a reply for each of them. The reply header includes the view number, the request identifier, *rid*, an MD5 digest of the operation result, the identifier of the replica, and the size of the result in bytes, *ressz*. Additionally, the reply message contains the operation result if the replica is the designated replier. The other replicas omit the result from the reply message and set the result size in the header to -1. Reply messages contain a single UMAC32 nonce and tag that authenticates the reply header. The client checks the MAC in the replies it receives and it also checks the result digest in the reply with the result.

Note that the MACs are computed only over the 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).

6.2 Interface

We implemented the algorithm as a library with a very simple interface (see Figure 6-2). Some components of the library run on clients and others at the replicas.

```
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);
```

Figure 6-2: The replication library API.

On the client side, the library provides a procedure to initialize the client using a configuration file, which contains the public keys and IP addresses of the replicas. The library also provides a procedure, *invoke*, that is called to cause an operation to be executed. This procedure carries out the client side of the protocol and returns the result when enough replicas have responded. The library also provides a split interface 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 non-deterministic 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 5.3.2.

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 a non-deterministic value for the request by making an upcall to the *nondet* procedure. The non-deterministic choice associated with a request is also passed as an argument to the *execute* upcall.

6.3 BFS: A Byzantine-Fault-tolerant File System

We implemented BFS, a Byzantine-fault-tolerant NFS [S^+85] service, using the replication library. BFS implements version 2 of the NFS protocol. Figure 6-3 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.

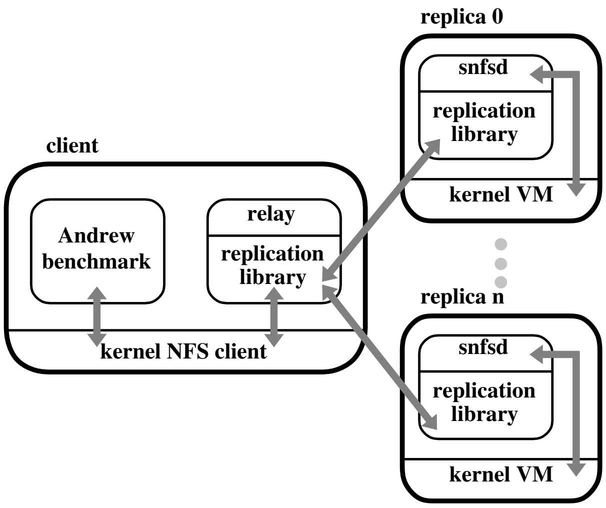

Figure 6-3: BFS: Replicated File System Architecture.

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

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

Our implementation ensures that all state machine replicas start in the same initial state and are deterministic, which are necessary conditions for the correctness of a service implemented using our protocol. The primary proposes the values for time-last-modified and time-last-accessed, and replicas select the larger of the proposed value and one greater than the maximum of all values selected for earlier requests. The primary selects these values by executing the upcall to compute non-deterministic 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 meta-data through replication as was done in Harp [LGG+91]. 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 [LGG+91].

Chapter 7

Performance Model

Analytic models are invaluable to explain the results of experiments and to predict performance in experimental conditions for which no measurements are performed. But care must be taken to ensure that they match reality. This chapter develops an analytic model for the performance of replicated services implemented using the BFT library. We validate the model by showing that it predicts the experimental results in the next chapter with accuracy. The model ignores the cost of checkpoint management, view changes, key refreshment, and recovery; these costs are analyzed in the next chapter.

7.1 Component Models

The experimental results show that the time to execute operations on a replicated service has three major components: digest computation, MAC computation, and communication.

7.1.1 Digest Computation

The model for the time to compute digests is simple. It has only two parameters: a fixed cost, D_f, and a cost per byte, D_v. The time to compute the digest of a string with l bytes is modeled as:

$TD(l) = D_f + D_v \times l$

This model is accurate for the MD5 [Riv92] cryptographic hash function, which is used in the current implementation of the BFT library. Another model parameter related to digest computation is the size of digests in bytes, *SD*.

7.1.2 MAC Computation

We intended to use a similar model for the time to compute MACs but our experimental results showed that such a model would be extremely inaccurate for small input strings. Instead, we measured the time to compute a MAC in microseconds, $TM(l)$, for each string size of l bytes. This was feasible because our current implementation only computes MACs on strings with one of two constant sizes (40 or 48 bytes).

The size of MACs in bytes is $SM = SMN + SMT$, where SMN is the size of the MAC nonce and SMT is the size of the MAC tag (both 8 bytes in UMAC32 [BHK$^+$99]).

Replies contain a single MAC but other messages contain authenticators. Authenticators have a MAC for each replica except that when the sender is a replica they do not have a MAC for the sender. Thus, the time to generate an authenticator TGA in microseconds is modeled as:

$TGA_c(l, n) = n \times TM(l)$, for a client or

$TGA_r(l, n) = (n - 1) \times TM(l)$, for a replica.

Here l is the size of the string the MAC is computed on and n is the number of replicas. The time to verify an authenticator is modeled as:

$TVA(l) = TM(l)$, for a client or a replica.

Since the library uses a single nonce for all the MACs in an authenticator, the size of an authenticator in bytes is given by the formula:

$SA_c(n) = n \times SMT + SMN$, for a client or

$SA_r(n) = (n - 1) \times SMT + SMN$, for a replica.

7.1.3 Communication

The performance model for communication assumes that each client and each replica is connected by a dedicated full-duplex link to a store-and-forward switch. All the links have the same bandwidth and the switch can forward both unicast and multicast traffic at link speed. The model assumes that the propagation delay on the cables connecting the hosts to the switch is negligible. The switch does not flood multicast traffic on all links; instead multicast traffic addressed to a group is only forwarded on the links of hosts that are group members. The model also assumes that messages are not lost; this is reasonable when the loss rate (due to congestion or other causes) is sufficiently low not to affect performance. These assumptions match our experimental environment, which is described in Section 8.1.

The first attempt to model the communication time used a fixed cost, C_f, and a cost per byte, C_v: the time to send a message with l bytes between two hosts was modeled as: $TC(l) = C_f + C_v \times l$. Unfortunately, this simple model does not separate the time spent at the hosts from the time spent in the switch. Therefore, it cannot predict the communication time with accuracy when multiple messages are sent in parallel or when a message is fragmented. To avoid this problem, we broke communication time into time spent in the switch, and time spent computing at each of the hosts.

The model for the time spent in the switch has two parameters: a fixed cost in microseconds, S_f, and a variable cost in microseconds per byte, S_v. The fixed cost is the switch latency and the variable cost is the inverse of the link bandwidth.

The actual time spent in the switch by a frame sent between hosts depends on the load on the switch. It always takes the switch $S_v \times l$ microseconds to receive all the bits in the frame. Since the switch is store-and-forward, it waits until it receives all the bits before forwarding the frame on an output link. Then, it takes an additional S_f microseconds before forwarding the frame. If the output links are free, it takes $S_v \times l$ microseconds to forward the frame. Otherwise, there is an additional delay while other frames are forwarded.

The model for the computation time at the hosts also has two parameters: H_f is a fixed cost in microseconds and H_v is the cost per byte. The computation time, $TH(l)$, to send a frame of l bytes is modeled as:

$TH(l) = H_f + H_v \times l$

The computation time to receive a frame of l bytes is assumed to be identical for simplicity. The accuracy of the model suggests that this is reasonable in our experimental environment.

Combining the two models yields the following total communication time for a frame of l bytes without congestion:

$TC(l) = S_f + 2S_v \times l + 2TH(l)$

When several messages are sent in parallel, it is necessary to reason how the computation times at the hosts and the switch overlap in order to compute the total communication time. For example, Figure 7-1 shows a time diagram for the case where n hosts send frames of l bytes in parallel to the same host. The communication time in this case is:

$TC_{par}(l, n) = 2TH(l) + S_f + 2S_v \times l + (n-1)max(S_v \times l, TH(l))$

It is necessary to take the maximum because the receiver can process frames only after it receives them but it may take longer for the receiver to process a frame than its transmission time.

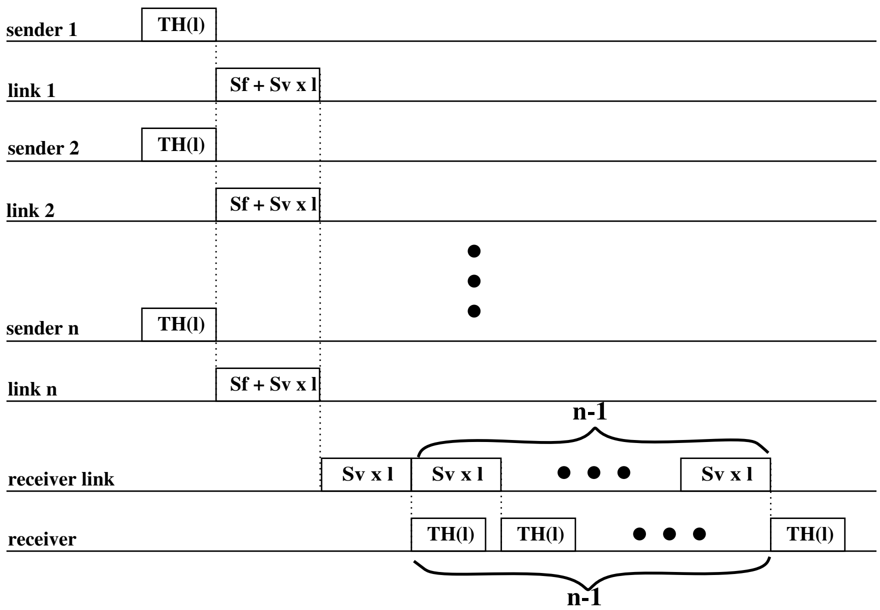

Figure 7-1: Performance model: time to send n frames with l bytes in parallel.

The model uses frames rather than messages to compute the communication time. To complete the model, it is necessary to define a mapping between the messages sent by the library and the frames actually sent on the wire. These differ because messages may be fragmented into several frames and frames include additional protocol headers and trailers. For example, IP fragments UDP messages sent over Ethernet when their size is greater than 1472 bytes. We define $NF(l)$ as the number of fragments for a message of l bytes. The message has $NF(l) - 1$ fragments whose frames have the maximum size, *MFS*, and one fragment that contains the remaining bytes. The function $RFS(l)$ returns the frame size of the fragment that contains the remaining bytes. The mapping between messages and frames is used next to derive an expression for the communication time of fragmented messages.

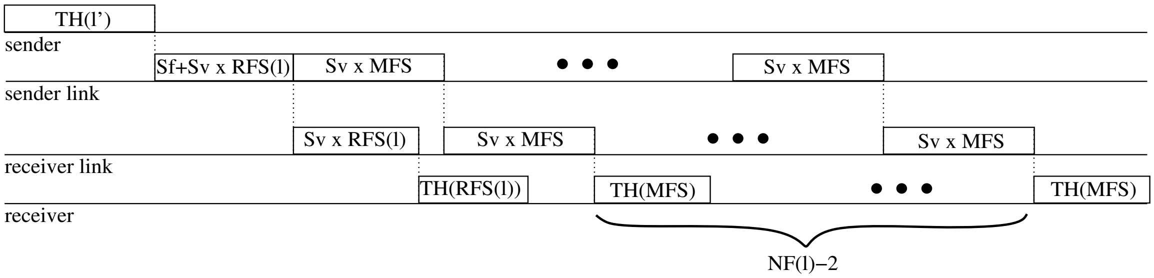

Figure 7-2: Performance model: time to send a message with l bytes that is fragmented. l' is the size of the message plus the number of bytes of protocol overhead ($l' = RFS(l) + (NF(l) - 1) \times MFS$).

Figure 7-2 shows a time diagram for the case where a host sends a message of l bytes that is fragmented. This figure assumes that the small fragment is sent first as it is done in the Linux kernel in our experimental setup. The figure also reflects the fact that in Linux the sender performs almost all the computation before the first fragment is sent on the wire. The communication time in this case is:

$$\begin{aligned} TC_{frag}(l) = {} & TH(RFS(l) + (NF(l) - 1) \times MFS) + S_f + 2S_v \times RFS(l) \\ & + max(S_v \times (2MFS - RFS(l)), TH(RFS(l))) \\ & + (NF(l) - 2) \times max(S_v \times MFS, TH(MFS)) + TH(MFS) \end{aligned}$$

7.2 Protocol Constants

Table 7.1 describes several constants that are characteristic of the protocol used by the BFT library and independent of the experimental environment. These constants appear in the analytic models for latency and throughput presented in the following sections.

7.3 Latency

We will now derive a model for the latency of the replicated service using the component models presented in the previous section. We will start with read-only operations because they are simpler.

name	value	description
RID	12 bytes	sum of the sizes of the client and request identifiers
REQH	40 bytes	size of request message header
REPH	48 bytes	size of reply message header
PPH	48 bytes	size of pre-prepare message header
PH	48 bytes	size of prepare message header

Table 7.1: Protocol Constants

7.3.1 Read-Only Operations

Figure 7-3 shows a timing diagram for a read-only operation. The client starts by digesting the operation argument, the client identifier, and the request identifier. Then, it places the resulting digest in the request header and computes an authenticator for the header that is appended to the request message. Next, the request is sent to all the replicas. The replicas check the authenticator and the digest. If the message passes these checks, the replicas execute the operation. The reply message includes a digest of the operation result in its header and a MAC of the header. After building the reply messages, the replicas send them to the client.

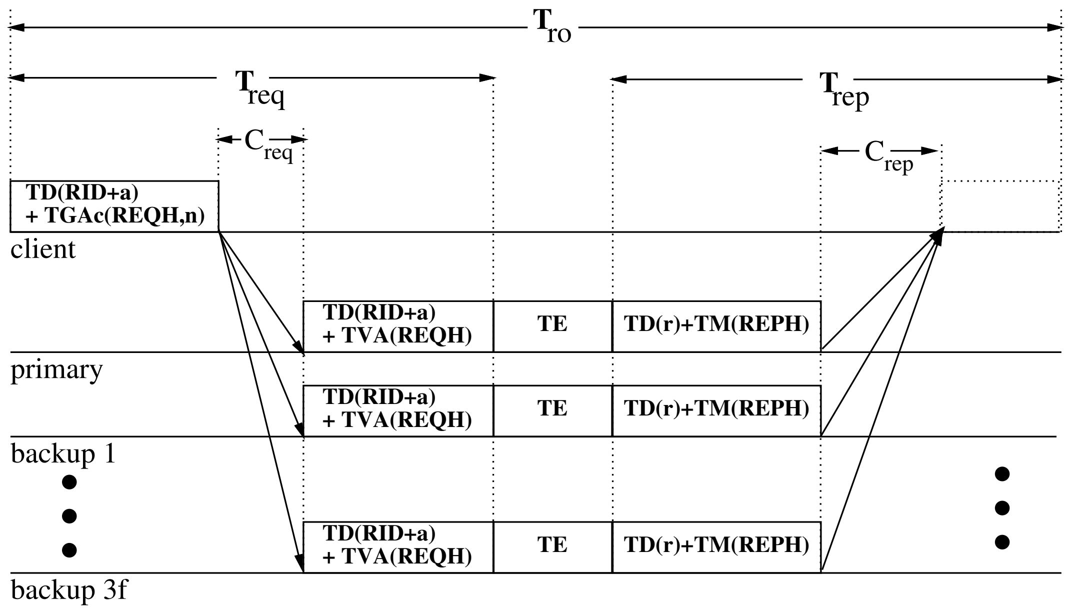

Figure 7-3: Performance model: read-only requests. Here, a is the size of the argument to the requested operation, r is the size of the operation result, and n is equal to $3f + 1$.

The total time to execute a request is the sum of the time T_{req} until a request is ready for execution at the replicas, the execution time *TE*, and the time T_{rep} from the execution of the request till the client receives enough replies.

$T_{ro}(a, r, n) = T_{req}(a, n) + TE + T_{rep}(r, n)$

$T_{req}(a, n) = 2TD(RID + a) + TGA_c(REQH, n) + TVA(REQH) + C_{req}(a, n)$

$T_{rep}(r, n) = TD(r) + TM(REPH) + C_{rep}(r, n)$

Here, a is the size of the operation argument, r is the size of the operation result, n is the number of replicas, and C_{req} and C_{rep} are the communication time for the request and the replies, respectively.

The communication time for the request depends on whether the request is fragmented or not. It is given by the formula:

$C_{req}(a, n) = TC(RFS(REQS(a, n)))$, if $NF(REQS(a, n)) = 1$

$TC_{frag}(REQS(a, n))$, otherwise.

with $REQS(a, n) = REQH + a + SA_c(n)$ (i.e., the request size).

The communication time for replies also depends on the size, r, of the operation result. There are three cases. Figure 7-4 depicts the first case where r is sufficiently large that digest replies are used but small enough that the reply with the operation result is not fragmented. The Figure assumes that the reply with the result is scheduled last on the client link. This overestimates the communication cost; latency may be lower if this reply is one of the first $2f + 1$ to be scheduled.

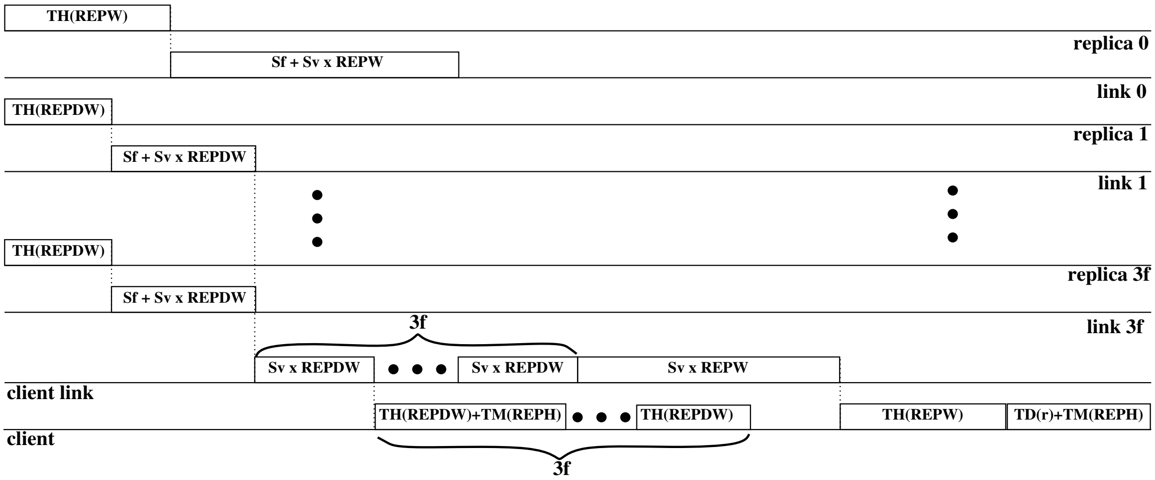

Figure 7-4: Communication time for replies that are not fragmented. *REPW* is the size of the reply frame with the result of the operation and *REPDW* is the size of a frame with a digest reply.

The communication time in this case is:

$C_1(r, n) = max(TH(REPW(r)) + S_f + S_v \times REPW(r), TH(REPDW) + S_f + (3f + 1)S_v \times REPDW)$

$C_2(r, n) = max(C_1(r, n) + S_v \times REPW(r),$

$TH(REPDW) + S_f + 2S_v \times REPDW + 3fTH(REPDW) + 2fTM(REPH))$

$C_{rep}(r, n) = C_2(r, n) + TH(REPW(r)) + TD(r) + TM(REPH)$

$REPW(r) = RFS(REPH + r + SM)$ is the size of the reply frame with the result of the operation, $REPDW = RFS(REPH + SM)$ is the size of a frame with a digest reply, C_1 is the time when the frame with the result starts being forwarded on the client link, and C_2 is the time when the client starts processing this frame. These formulas account for the overlap between the time to verify the

MACs in replies and communication.

In the second case, the reply message with the result is fragmented. To derive the formula for C_{rep} in this case, we combine the last formula with the formula for TC_{frag}. We assume that the time between the instants the first bit of the first fragment and the last bit of the last fragment are forwarded on the client link is $S_v \times NF(REPH + r + SM) \times MFS$. This was always true in Figure 7-2 but here the time may be smaller if congestion due to the other replies delays forwarding for sufficiently long (this only happens for $f \geq 6$ in our experimental setup).

The value of C_{rep} with fragmentation is given by the following formulas:

$$\begin{aligned}
C_3(r,n) = {} & max(TH(RFS(REPS(r)) + (NF(REPS(r)) - 1)MFS) + S_f + S_v \times RFS(REPS(r)), \\
& TH(REPDW) + S_f + (3f+1)S_v \times REPDW) \\
C_4(r,n) = {} & max(C_3(r,n) + S_v \times RFS(REPS(r)), \\
& TH(REPDW) + S_f + 2S_v \times REPDW + 3fTH(REPDW) + 2fTM(REPH)) \\
C_5(r,n) = {} & max(C_4(r,n) + TH(RFS(REPS(r))) + (NF(REPS(r)) - 2)TH(MFS), \\
& C_3(r,n) + S_v \times NF(REPS(r)) \times MFS)) \\
C_{rep}(r,n) = {} & C_5(r,n) + TH(MFS) + TD(r) + TM(REPH)
\end{aligned}$$

Here, $REPS(r) = REPH + r + SM$, C_3 is the time when the first fragment starts to be forwarded on the client link, C_4 is the time when the client starts to process the first fragment, and C_5 is the time when the client starts to process the last fragment.

The third case occurs when r is less than a threshold (currently 33 bytes). In this case, all replicas send replies with the operation result instead of using the digest replies optimization. Since all replies have the same size and are not fragmented, we use the formula for TC_{par} modified to account for the overlap between MAC computation and communication. The value of C_{rep} is:

$$\begin{aligned}
C_{rep}(r,n) = {} & 2TH(REPW(r)) + S_f + 2S_v \times REPW(r) \\
& +2f \times max(S_v \times REPW(r), TH(REPW) + TM(REPH)) + TD(r)
\end{aligned}$$

7.3.2 Read-Write Operations

Next, we derive a model for read-write operations. There are two cases depending on the size of the operation argument. If the size of the argument is less than a threshold (currently 256 bytes), the client sends the request only to the primary and the request is inlined in the pre-prepare message. Otherwise, the client multicasts the request to all replicas and the pre-prepare message includes only the digest of the request. Figure 7-5 shows a time diagram for the second case.

The first part of the read-write algorithm is identical to the read-only case. Thus, T_{req} can be computed using the same formula. After checking the request, the primary computes the digest of the digest in the request header. Then, it constructs a pre-prepare message with the resulting digest in its header and an authenticator for the header. The backups check the pre-prepare message by verifying the authenticator and recomputing the digest. If they accept the pre-prepare and already have a matching request, they build a prepare message with an authenticator and send it to all other

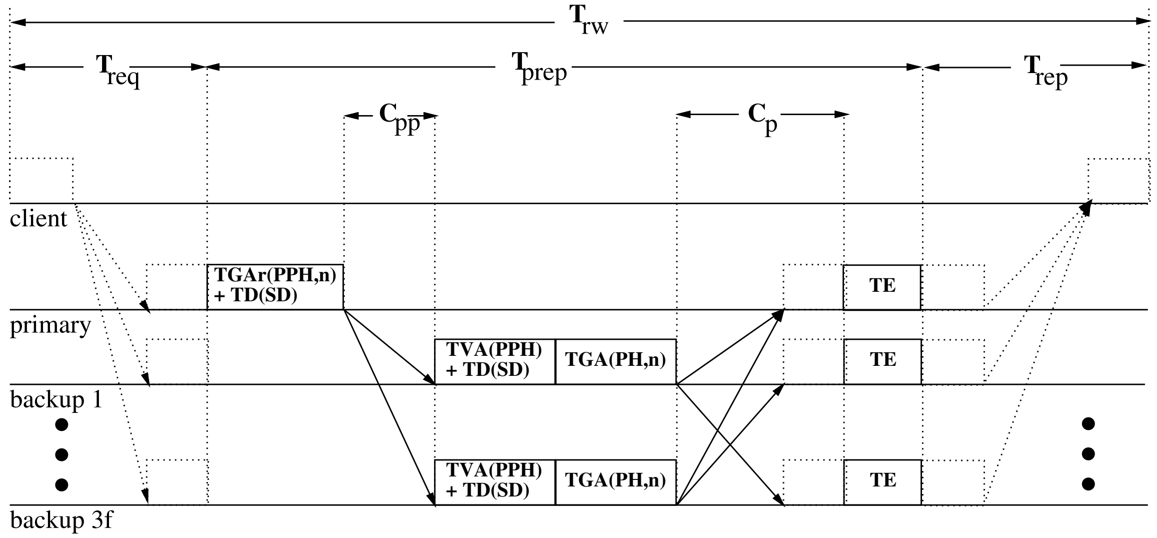

Figure 7-5: Performance model: read-write requests.

replicas. After replicas have prepared the request, they execute it and the algorithm proceeds as in the read-only case; T_{rep} is given by the same formulas.

The total time to execute the read-write request in the figure is the sum of T_{req}, the time T_{prep} from the moment the primary starts to build the prepare message till the request is prepared, the execution time *TE*, and T_{rep}:

$$T_{rw}(a,r,n) = T_{req}(a,n) + T_{prep}(a,n) + TE + T_{rep}(r,n)$$
$$T_{prep}(a,n) = 2TD(SD) + TGA_r(PPH,n) + TVA(PPH,n)$$
$$+TGA_r(PH,n) + C_{pp}(a,n) + C_p(n)$$

The communication time for the pre-prepare message, $C_{pp}(a,n)$, is computed using a formula similar to C_{req}; it is:

$$C_{pp}(a,n) = TC(RFS(PPS(a,n))), \text{ if } NF(PPS(a,n)) = 1$$
$$TC_{frag}(PPS(a,n)), \text{ otherwise.}$$

with $PPS(a,n) = PPH + SD + SA_r(n)$ (i.e., the pre-prepare size).

The communication time for prepare messages is similar in structure to TC_{par} but it accounts for the overlap between authenticator verification and computation:

$$C_p(n) = 2TH(PW(n)) + S_f + 2S_v \times PW(n)$$
$$+max((3f-1)(S_v \times PW(n)), (3f-1)TH(PW(n)) + (2f-1)TVA(PH)) + TVA(PH)$$

with $PW(n) = RFS(PH + SA_r(n))$ (i.e., the prepare size on the wire).

The case when requests are inlined in the pre-prepare message is similar. The differences are that C_{pp} increases because the pre-prepare message is bigger and that backups only check the request when they receive the pre-prepare message. The resulting formulas are:

$$T_{prep}(a,n) = 2TD(SD) + TGA_r(PPH,n) + TVA(PPH,n) + TD(RID + a) + TVA(REQH)$$
$$+TGA_r(PH,n) + C_{pp}(a,n) + C_p(n)$$

$$C_{pp}(a,n) = TC(RFS(PPS(a,n))), \text{ if } NF(PPS(a,n)) = 1$$
$$TC_{frag}(PPS(a,n)), \text{ otherwise.}$$
with $PPS(a,n) = PPH + REQS(a,n) + SA_r(n)$

7.4 Throughput

We obtain a model for the throughput of a replicated system by developing a model for the time to process a batch of requests. This model is based on the latency models in the previous section but it has two additional parameters: the batch size b and the number of client machines m. Each client sends b/m of the requests in the batch. For simplicity, we assume that all the clients send the requests at the same time.

7.4.1 Read-Only Requests

We start with read-only requests again because they are simpler. The strategy is to split the total time, T^b_{ro}, into the sum of two components: the time to get the requests ready to execute at the replicas, T^b_{req}, and the time to execute the requests and get the replies to the clients, T^b_{erep}. The value of each of these components is obtained by taking the maximum of the computation times over all the nodes and the communication times over all the links. An accurate model for latency requires careful reasoning about scheduling of communication and computation at the different components but taking the maximum is a good approximation for large request batches.

We use Figure 7-3 and the formulas for T_{req} in the previous section to derive the following formulas for T^b_{req}:

$$T^b_{req_c}(a,n,b,m) = b \times (TD(RID + a) + TGA_c(REQH, n) + TH(REQW(a,n)))/m$$
$$T^b_{req_r}(a,n,b,m) = b \times (TD(RID + a) + TVA(REQH) + TH(RFS(REQS(a,n))) + (NF(REQS(a,n)) - 1)TH(MFS))$$
$$T^b_{req_{cl}}(a,n,b,m) = b \times S_v \times REQW(a,n)/m$$
$$T^b_{req_{rl}}(a,n,b,m) = b \times S_v \times REQW(a,n)$$
$$T^b_{req}(a,n,B,m) = max(T^b_{req_c}(a,n,b,m), T^b_{req_r}(a,n,b,m), T^b_{req_{cl}}(a,n,b,m), T^b_{req_{rl}}(a,n,b,m))$$
with $REQW(a,n) = RFS(REQS(a,n)) + (NF(REQS(a,n)) - 1) \times MFS.$

Here, *REQW* is the number of bytes in frames that contain the request. $T^b_{req_c}$ is the computation time at each client; it is equal to the corresponding client computation time for a single request multiplied by b/m (because each client sends only b/m requests). Replicas receive all the requests in the batch so their computation time is multiplied by b; this is reflected in the formula for the computation time at each replica, $T^b_{req_r}$. Similarly only b/m requests flow over each client link whereas b requests go through each replica's link. This is accounted for in the formulas for $T^b_{req_{cl}}$, which is the communication time at each client link, and $T^b_{req_{rl}}$, which is the communication time at each replica link.

T^b_{erep} can be computed using the following formulas (ignoring the case without digest replies to simplify the model):

$$
\begin{aligned}
T^b_{erep_c}(r,n,b,m) &= b \times (\mathit{TD}(r) + (2f+1)\mathit{TM}(\mathit{REPH}) + 3f \times \mathit{TH}(\mathit{REPDW}) \\
&\quad + \mathit{TH}(\mathit{RFS}(\mathit{REPS}(r))) + (\mathit{NF}(\mathit{REPS}(r)) - 1)\mathit{TH}(\mathit{MFS}))/m \\
T^b_{erep_r}(r,n,b,m) &= b \times (\mathit{TE} + \mathit{TD}(r) + \mathit{TM}(\mathit{REPH})) + \mathit{TH}(\mathit{REPW}(r))b/n + \mathit{TH}(\mathit{REPDW})(b - b/n) \\
T^b_{erep_{cl}}(r,n,b,m) &= b \times S_v \times (\mathit{REPW}(r) + 3f \times \mathit{REPDW})/m \\
T^b_{erep_{rl}}(r,n,b,m) &= S_v \times (REPW(r) \times b/n + REPDW \times (b - b/n)) \\
T^b_{erep}(r,n,B,m) &= \mathit{max}(T^b_{erep_c}(r,n,b,m), T^b_{erep_r}(r,n,b,m), T^b_{erep_{cl}}(r,n,b,m), T^b_{erep_{rl}}(r,n,b,m))
\end{aligned}
$$

$\mathit{REPW}(r)$ and REPDW were defined previously; they are the number of bytes in frames with the operation result and the number of bytes in frames with digest replies, respectively. $T^b_{erep_c}$ is the computation time at each client; it accounts for receiving $3f+1$ replies, computing the result digest, and authenticating $2f+1$ replies for each of the b/m requests sent by a client. Each replica executes b requests and computes a result digest and a MAC for the reply to each of them. But a replica only sends b/n replies with the operation result; the other replies contain only digests. This is reflected in the formula for $T^b_{erep_r}$, which is the computation time at each replica. $T^b_{erep_{cl}}$ is the communication time at each client's link, and $T^b_{erep_{rl}}$ is the communication time at each replica's link.

Using these formulas, we can now compute the time to execute the batch of read-only requests:

$T^b_{ro}(a,r,n,b,m) = T^b_{req}(a,n,b,m) + T^b_{erep}(r,n,b,m)$

The throughput in operations per microsecond is $b/T^b_{ro}(a,r,n,b,m)$.

7.4.2 Read-Write Requests

The time to execute a batch of read-write requests is split into the sum of three components: T^b_{req}, T^b_{erep}, and the time for the batch of requests to prepare, T^b_{prep}. T^b_{req} and T^b_{erep} can be computed using the formulas derived for read-only requests. The formula for T^b_{prep} is identical to the formula for T_{prep} except that it accounts for the fact that the pre-prepare message is sent for a batch of requests. In the case where, requests are inlined in the pre-prepare message T^b_{prep} is:

$$
\begin{aligned}
T^b_{prep}(a,n,b) &= b \times (\mathit{TD}(\mathit{RID}+a) + 2\mathit{TD}(\mathit{SD}) + \mathit{TVA}(\mathit{REQH})) \\
&\quad + \mathit{TGA}_r(\mathit{PPH},n) + \mathit{TVA}(\mathit{PPH},n) + \mathit{TGA}_r(\mathit{PH},n) + C^b_{pp}(a,n,b) + C_p(n) \\
C^b_{pp}(a,n,b) &= \mathit{TC}(\mathit{RFS}(\mathit{PPS}^b(a,n,b))), \text{ if } \mathit{NF}(\mathit{PPS}(a,n)) = 1 \\
&\quad\; \mathit{TC}_{frag}(\mathit{PPS}^b(a,n,b)), \text{ otherwise.} \\
\mathit{PPS}^b(a,n,b) &= \mathit{PPH} + b \times \mathit{REQS}(a,n) + \mathit{SA}_r(n)
\end{aligned}
$$

Here, $\mathit{PPS}^b(a,n,b)$ is the size of a pre-prepare message with b copies of requests for an operation with argument size a; and C^b_{pp} is the communication time for the message, which is identical to C_{pp} except that the pre-prepare message is larger.

There are two differences when the requests are not inlined in the pre-prepare message: the size of this message decreases because it includes only digests of the requests rather than copies; and the backups check the requests in parallel with the primary, which eliminates $b \times (\mathit{TD}(\mathit{RID} + a) + \mathit{TVA}(\mathit{REQH}))\mu$s. This is reflected in the following formulas for T^b_{prep} when requests are not inlined:

$$
\begin{aligned}
T^b_{prep}(a, n, b) &= 2b \times \mathit{TD}(\mathit{SD}) \\
&\quad + \mathit{TGA}_r(\mathit{PPH}, n) + \mathit{TVA}(\mathit{PPH}, n) + \mathit{TGA}_r(\mathit{PH}, n) + C^b_{pp}(a, n, b) + C_p(n) \\
\mathit{PPS}^b(a, n, b) &= \mathit{PPH} + b \times \mathit{SD} + \mathit{SA}_r(n)
\end{aligned}
$$

These formulas allow us to compute the time to execute the batch of read-write requests:
$T^b_{rw}(a, r, n, b, m) = T^b_{req}(a, n, b, m) + T^b_{prep}(a, n, b) + T^b_{erep}(r, n, b, m)$
The throughput in operations per microsecond is $b/T^b_{rw}(a, r, n, b, m)$.

7.5 Discussion

The analytic model for latency has some properties that are worth highlighting:

- T_{req} grows linearly with the number of replicas because of authenticator generation and increased communication cost due to growth in the size of request authenticators. T_{req} grows linearly with the argument size due to increased communication and digest computation time for requests.
- T_{rep} grows linearly with the number of replicas because each replica sends a reply to the client. T_{rep} also grows linearly with the result size due to increased communication and digest computation time for replies.
- T_{prep} is (mostly) independent of argument and result sizes. However, it grows with the square of the number of replicas because of the prepare messages that are sent in parallel by the backups and contain authenticators whose size grows linearly with the number of replicas.
- The overhead introduced by adding additional replicas is (mostly) independent of operation argument and result sizes.

The same observations are valid for the corresponding components in the throughput model. According to this model, the only cost that grows with the square of the number of replicas, T^b_{prep}, is amortized over the batch size. Additionally, the computation time at a replica and the communication time in its link decrease linearly with the number of replicas (if there are more clients than replicas).

Chapter 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 chapter presents results of experiments to evaluate the performance of these systems. The results show that they perform well — systems implemented with the BFT library have performance that is competitive with unreplicated 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 micro-benchmarks to evaluate the performance of the replication library in a service-independent way and to determine the impact of each of our optimizations. We also measured performance when the number of replicas increases and we used the analytic model to study sensitivity to variations in the model parameters.

The experiments were performed using the setup in Section 8.1. We describe experiments to measure the value of the analytic model parameters in Section 8.2. Section 8.3 uses micro-benchmarks to evaluate the performance during the normal case without checkpoint management, view changes, key refreshment, or recovery. Sections 8.4 and 8.5 present results of experiments to evaluate the performance of checkpoint management, and view changes, respectively. Section 8.6 studies the performance of the BFS file system with and without proactive recoveries.

The main results in this chapter are summarized in Section 8.7.

8.1 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 10K 18WLS 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. Each machine was connected by a single Category 5 cable to a full-duplex port in an Extreme

Networks Summit48 V4.1 switch. This is a store-and-forward switch that can forward IP unicast and multicast traffic at link speed. Additionally, it performs IGMP snooping such that multicast traffic is forwarded only to the members of the destination group. All experiments ran on an isolated network and we used the Pentium cycle counter to measure time accurately.

The library was configured as follows. The checkpoint period, K, was 128 sequence numbers, which causes garbage collection to occur several times in each experiment. The size of the log, L, was 256 sequence numbers. The state partition tree had 4 levels, each internal node had 256 children, and the leaves had 4 KB. Requests for operations with argument size greater than 255 bytes were transmitted separately; the others were inlined in pre-prepares. The digest replies optimization was not applied when the size of the operation result was less than or equal to 32 bytes. The window size for request batching was set to 1.

8.2 Performance Model Parameters

In order to use the analytic model to explain the experimental results in the next sections, it is necessary to measure the value of each parameter in the model in our experimental setup. This section describes experiments to measure these values.

8.2.1 Digest Computation

The BFT library uses the MD5 [Riv92] cryptographic hash function to compute digests. We ran an experiment to measure the time to compute MD5 digests as a function of the input string. The experiment was designed such that the input string was not in any of the processor caches before being digested. Figure 8-1 presents the results.

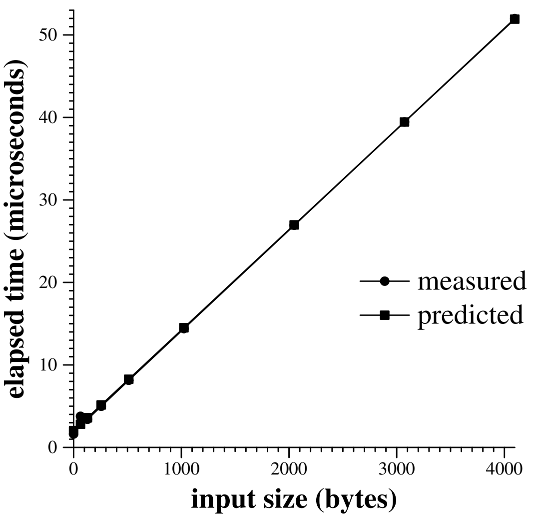

Figure 8-1: Time to compute MD5 digests as a function of the input size.

We used a linear regression (least squares method) to compute the parameters D_f and D_v in the digest computation model. Table 8.1 shows the values we obtained and Figure 8-1 shows digest computation times predicted with $TD(l) = D_f + D_v \times l$. The predicted and measured values are almost indistinguishable as evidenced by a high coefficient of determination (0.999).

parameter	value	description
D_f	2.034 μs	time to digest 0 bytes
D_v	0.012 μs/byte	additional cost per byte
SD	16 bytes	digest size

Table 8.1: Digest computation model: parameter values

8.2.2 MAC Computation

The BFT library only computes MACs of message headers that have a constant size of either 40 or 48 bytes. We ran an experiment to measure the time to compute these MACs using the UMAC32 [BHK+99] algorithm. The parameter values for the model are listed in Table 8.2.

parameter	value	description
$TM(40)$	965 ns	time to MAC 40 bytes
$TM(48)$	958 ns	time to MAC 48 bytes
SMT	8 bytes	size of MAC tag
SMN	8 bytes	size of MAC nonce

Table 8.2: MAC computation model: parameter values

8.2.3 Communication

The communication model is split into two components: time spent at the switch and time spent at the hosts. To separate out these two components, we measured round-trip latency for different frame sizes with and without the switch. In the configuration without the switch, the two hosts were connected directly by a crossover Category 5 cable.

According to our model, the total (one-way) communication time through the switch for a frame of l bytes without congestion is:

$$TC(l) = S_f + 2S_v \times l + 2TH(l)$$

The same communication time without the switch is:

$$TC^{ns}(l) = S_v \times l + 2TH(l)$$

Therefore, the difference between the measured round-trip times is:

$$\delta(l) = 2(TC(l) - TC^{ns}(l)) = 2(S_f + S_v \times l)$$

The reasoning assumes that the propagation delay on the network cables is negligible. This is a good assumption in our experimental environment; we use only Category 5 cables that add a maximum delay of 0.011μs per meter [Spu00] and our cables are significantly shorter than 10 meters.

We ran a linear regression with the values $\overline{\delta}(l)/2$ obtained by dividing the difference between the measured round-trip times by two. It yielded the values $S_f = 9.79\mu$s and $S_v = 0.08\mu$s/B with a coefficient of determination of 0.999. The high coefficient of determination shows that the model matches the experimental data and $S_v = 0.08\mu$s/B also matches the nominal bandwidth of Fast Ethernet.

With the value of S_v, we computed $\overline{TH}(l)$ by subtracting $S_v \times l$ from the round-trip time measured without the switch and dividing the result by two. Finally, we performed a linear regression analysis on these values and obtained $H_f = 20.83\mu$s and $H_v = 0.011\mu$s/B with a coefficient of determination of 0.996. Table 8.3 shows the values of the parameters associated with the communication model.

parameter	value	description
S_f	9.79μs	switch latency
S_v	0.08μs/byte	inverse of link bandwidth
H_f	20.83μs	host time to send 0 byte frame
H_v	0.011μs/byte	host time to send each additional byte
MFS	1514 bytes	maximum size of frame with fragment

Table 8.3: Communication model: parameter values

To complete the communication model, it is necessary to define the functions that map between messages and frames. These functions have the following values in UDP/IP over Ethernet:

NF$(l) = 1$, if $l \leq 1472$
$1 + \lceil (l - 1472)/1480 \rceil$, otherwise

RFS$(l) = l + 42$, if $l \leq 1472$
$(l - 1472)$ mod $1480 + 34$, otherwise

The IP, UDP, and Ethernet headers and the Ethernet trailer sum 42 bytes in length. The maximum size for a frame is 1514 bytes. The fragment with the first bytes in the message has both IP and UDP headers so it can hold 1472 message bytes. The other fragments do not have the UDP header so they can hold up to 1480 message bytes.

We validated the communication model by comparing predicted and measured communication times for various message sizes. Figure 8-2 shows both absolute times and the relative error of the predicted values. The predicted values were obtained using: $TC(RFS(l))$ for messages that are not fragmented and $TC_{frag}(l)$ with fragmentation (these formulas are defined in Section 7.1.3). The model is very accurate; it deviates at most 3.6% from the measured values and all the points except the first have an error with absolute value less than 1%.

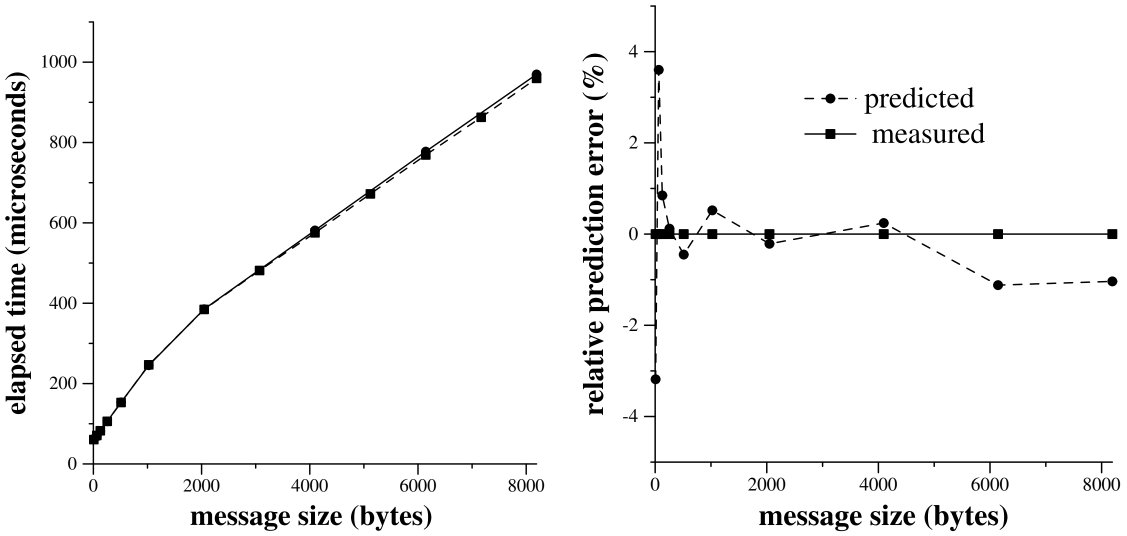

Figure 8-2: Communication time: measured and predicted values.

8.3 Normal Case

This section evaluates the performance during the normal case: there are no view changes or recoveries, and MAC keys are not refreshed. It compares 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.

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. These experiments provide a service-independent evaluation of the performance of the replication library.

Sections 8.3.1 and 8.3.2 describe experiments to evaluate the latency and throughput of the simple replicated service, respectively. Section 8.3.3 evaluates the impact of the various optimizations on performance. All these experiments use four replicas. In Section 8.3.4, we investigate the impact on performance as the number of replicas increases. Finally, Section 8.3.5 uses the analytic model to predict performance in a WAN environment and in a very fast LAN.

8.3.1 Latency

We measured the latency to invoke an operation when the service is accessed by a single client. All experiments ran with four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. 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.

Varying Argument Size

Figure 8-3 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. It has one graph with elapsed times and another with the slowdown of BFT relative to NO-REP. The graphs have results for both read-write and read-only operations.

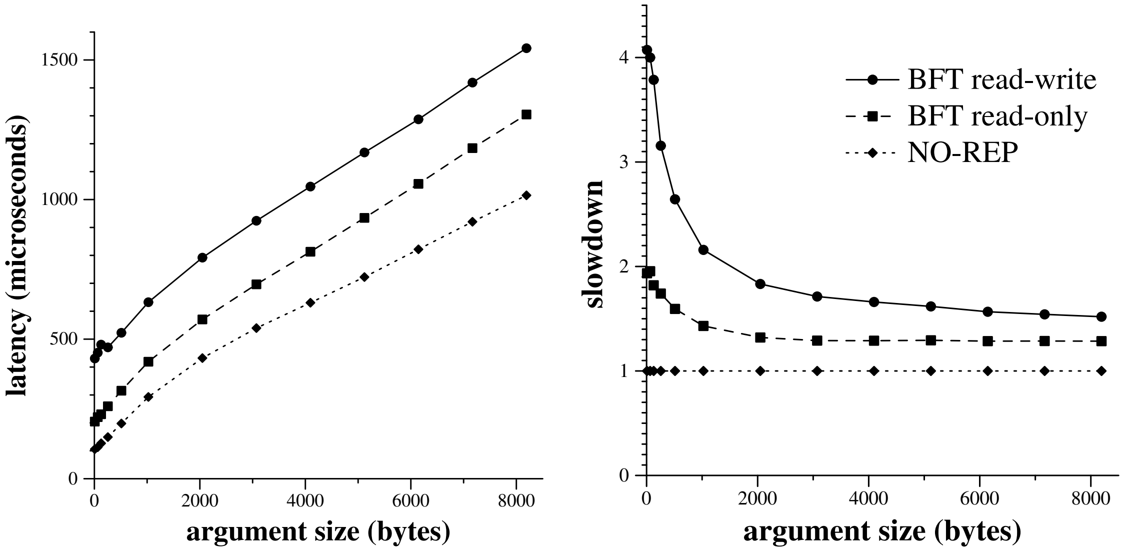

Figure 8-3: Latency with varying argument sizes: absolute times and slowdown relative to NO-REP.

The results show that the BFT library introduces a significant overhead relative to NO-REP in this benchmark. 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 (as shown in Section 8.6). The two major sources of overhead are digest computation and the additional communication due to the replication protocol. The cost of MAC computation is almost negligible (less than 3%).

The results show two major trends: the read-only optimization is very effective at reducing the slowdown introduced by the BFT library; and the slowdown decreases significantly as the size of the operation argument increases.

The read-only optimization improves performance by eliminating the time to prepare the requests. The analytic model predicts that this time does not change as the argument size increases (for arguments greater than 255 bytes). This is confirmed by the experimental results: the difference between the latency of read-only and read-write operations for the same argument size is approximately constant and equal to 225μs. Therefore, the speed up afforded by the read-only optimization decreases to zero as the argument size increases: it reduces latency by 52% with 8 B arguments but only by 15% for 8 KB arguments.

The slowdown for the read-write operation decreases from 4.07 with 8 B arguments to 1.52 with 8 KB arguments and it decreases from 1.93 to 1.29 with the read-only optimization. The decreased slowdown is also explained by the analytic model. The only component that changes as the argument size increases is T_{req}, which is the time to get the request to the replicas. T_{req} increases because the communication time and the time to digest the request grow with the argument size. In our experimental setup, the communication time increases faster than the digest computation time: communication increases $0.011 + 0.08 = 0.091\mu$s per byte (the sum accounts for the variable cost at the sender and at the switch); and the digest computation time increases $2 \times 0.012\mu$s per byte (which accounts for the variable cost of computing the request digest at both the client and the replicas). Since the communication cost of NO-REP also increases 0.091μs/byte, the model predicts that the slowdown will decrease as the argument size increases till an asymptote of $(0.091 + 2 \times 0.012)/0.091 = 1.26$, which is close to the experimental results for the read-only operation.

The performance model can predict the results in Figure 8-3 with very high accuracy. Figure 8-4 shows the error of the latency values predicted by the model relative to the values measured. The absolute value of the error is always below 2.3%.

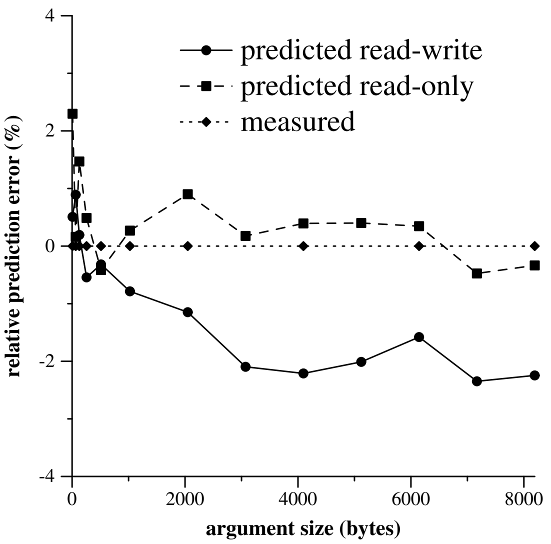

Figure 8-4: Latency model: relative prediction error for varying argument sizes.

Varying Result Sizes

Figure 8-5 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. The graphs in this figure are very similar to the ones for varying argument size: they also show that the read-only optimization is effective at reducing the slowdown introduced by the BFT library; and that the slowdown decreases significantly as the size of the operation result increases. The major sources of overhead are again additional communication and digest computation (this time for replies).

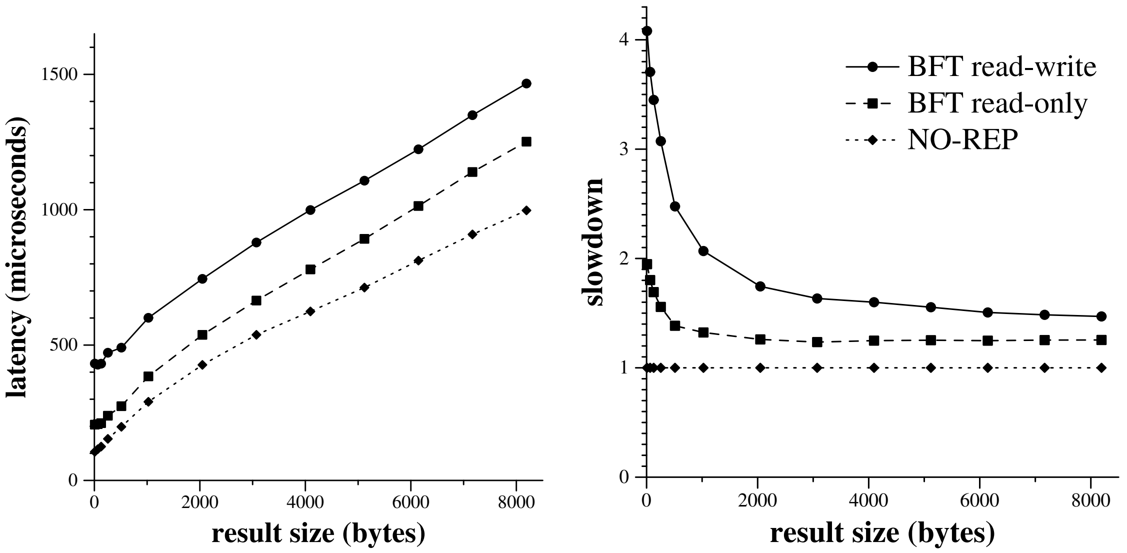

Figure 8-5: Latency with varying result sizes: absolute times and slowdown relative to NO-REP.

The impact of the read-only optimization can be explained exactly as before. In this case, the difference between the latency of read-only and read-write operations for the same result size is approximately constant and equal to 215μs. The optimization also speeds up latency by 52% with 8 byte results but only by 15% for 8 KB results.

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 argument why the slowdown decreases is similar to the one presented for varying arguments. But, in this case, the only component that changes as the result size increases is T_{rep}, which is the time to get the replies to the client. T_{rep} grows as the result size increases due to the increased communication cost to send the reply with the result to the client and due to the increased cost to compute the digest of the result at the replicas and the client. Since the communication cost in NO-REP increases at the same rate, the model predicts that the slowdown will decrease as the result size increases towards the same asymptote as before (1.26); this prediction is close to the experimental results.

The performance model can also predict latency with varying result sizes accurately. Figure 8-4 shows the error of the latency values predicted by the model relative to the values measured. The absolute value of the error is always below 2.7% for all result sizes except for 64 and 128 bytes, where it is as high as 11.5%. It is not clear why the model overestimates the latency for these result sizes but it may be due to our pessimistic assumption that the reply with the complete result is always scheduled last for forwarding on the client's link.

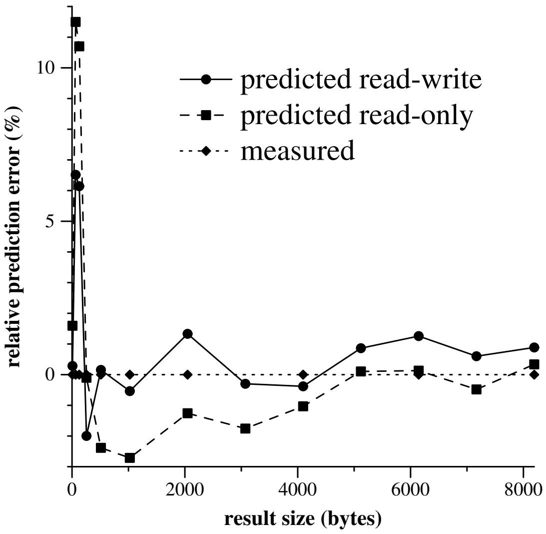

Figure 8-6: Latency model: relative prediction error for varying result sizes.

8.3.2 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 5 client machines[1] and each client process invoked operations synchronously, i.e., it waited for a reply before invoking a new operation. 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 8-7 shows throughput results for operation 0/0. The standard deviation was always below 2% of the reported values. 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

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

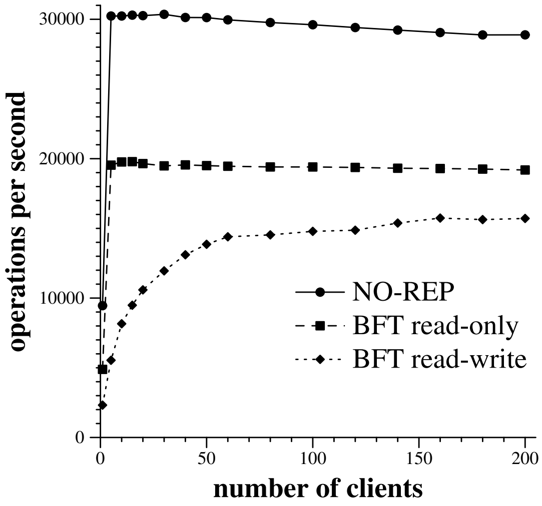

Figure 8-7: Throughput for operation 0/0 (with 8 byte argument and result).

because the cost of preparing the batch of requests is amortized over the size of the batch. In the current implementation, the size of the batch is limited by how many requests can be inlined in a pre-prepare message; this limit is equal to 101 requests for this operation. The average batch size in this experiment is approximately equal to the total number of clients divided by two (with the constraint that it is not greater than 101 requests). Therefore, the throughput of the read-write operation increases as the client population grows up to 200 and then it saturates.

Figure 8-8 shows throughput results for operation 0/4. Each point is an average of five independent runs for the read-write operation and ten for the read-only operation. The standard deviation was below 4% of the reported values for the read-write operation but was as high as 18% for the read-only operation.

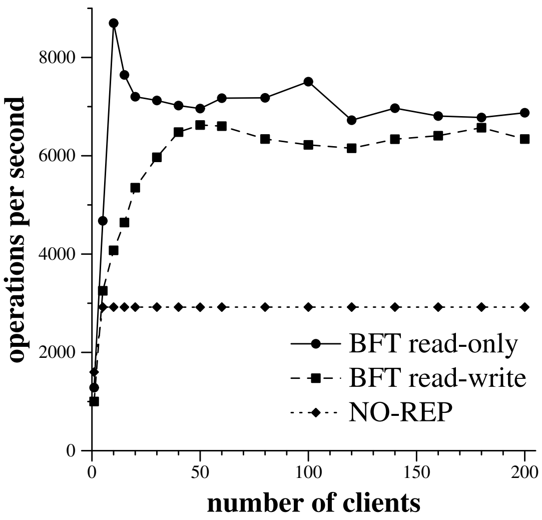

Figure 8-8: Throughput for operation 0/4 (with 8 byte argument and 4 KByte result).

BFT has better throughput than NO-REP. The bottleneck for NO-REP in operation 0/4 is the link bandwidth; NO-REP executes approximately 3000 operations per second, which saturates the link bandwidth of 12 MB/s. BFT achieves better throughput because of the digest-replies optimization: each client chooses one replica randomly; this replica's reply includes the 4 KB result but the replies of the other replicas only contain small digests. As a result, clients obtain the large replies in parallel from different replicas. BFT achieves a maximum throughput of 6625 operations per second for the read-write operation and 8698 operations per second with the read-only operation; this corresponds to an aggregate throughput of 26MB/s and 34 MB/s. The bottleneck for BFT is the replicas' CPU.

The throughput of the read-write operation increases with the number of clients because the cost of preparing the batch of requests is amortized over the batch size. The throughput with the read-only optimization is very unstable. The instability occurs because the system is not always fair to all clients; this results in a large variance in the maximum time to complete the K operations, which is the time we use to compute the throughput. The average time for the clients to compute the K operations remains stable. Figure 8-9 compares the throughput for this operation computed both using the maximum time and the average time to complete the K operations at all clients.

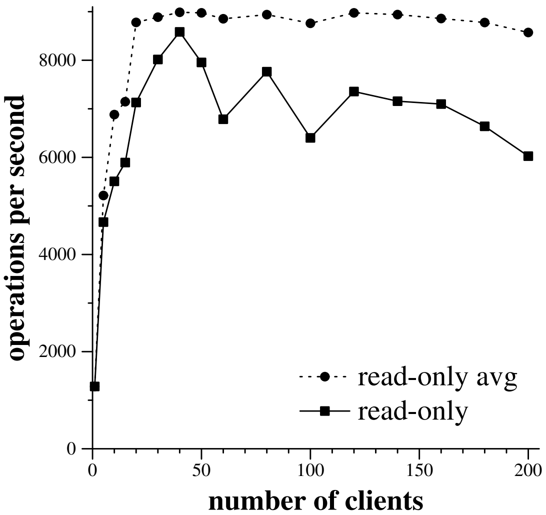

Figure 8-9: Throughput for read-only operation 0/4. The results labeled *avg* are based on the average time to complete the middle K operations rather than the maximum.

Figure 8-10 shows throughput results for operation 0/4. The standard deviation was below 7% of the reported value. There are no points with more than 15 clients for NO-REP operation 4/0 because of lost request messages; NO-REP uses UDP directly and does not retransmit requests.

The 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 3000 operations per second. NO-REP achieves a maximum throughput of 2921 operations per second while BFT achieves 2591 for read-write operations (11% less than NO-REP) and 2865 with the read-only optimization (2% less than NO-REP).

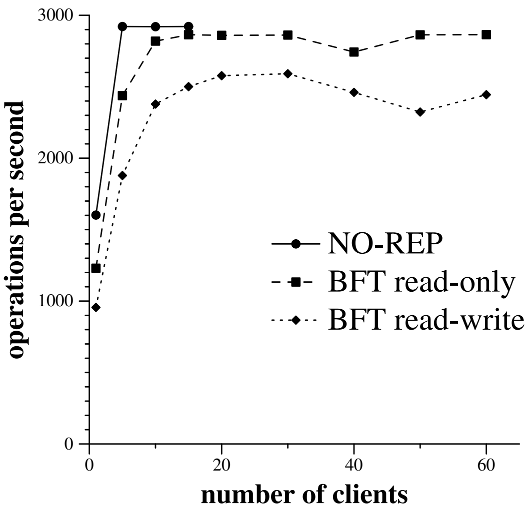

Figure 8-10: Throughput for operation 4/0 (with 4 KByte argument and 8 byte result).

Batching is once more responsible for increasing the throughput of the read-write operation as the number of clients increases. The requests for operation 4/0 are not inlined in pre-prepare messages and the current implementation imposes a limit of 16 such requests per batch. We measured an average batch size equal to the number of clients divided by two (up to the 16 request maximum). This explains why the throughput stops growing with approximately 30 clients. The throughput drops and its variance increases for more clients due to an increase in lost messages and retransmissions. This variance also disappears if we use the average time to complete the K operations to compute throughput rather than the maximum.

configuration	0/0	0/4	4/0
read-only	19707 (-0.4%)	8132 (-7%)	2717 (-5%)
read-write	14298 (-9%)	7034 (+6%)	2590 (0%)

Table 8.4: Throughput model: predicted values and errors relative to measured values.

The throughput performance model is accurate. Table 8.4 shows the maximum throughput values predicted by the model and the error relative to the values measured. The values for operations 0/0 and 0/4 were computed with a batch size of 101 and the values for operation 4/0 were computed with a batch size of 16. The absolute value of the error is always below 10%.

8.3.3 Impact of Optimizations

The experiments in the previous sections show that the read-only optimization is effective at reducing latency and improving throughput of services replicated using the BFT library. The read-only optimization is special because it can only be applied to operations that satisfy a specific semantic constraint (namely not modifying the state). This section analyses the performance impact of the other optimizations that are applied to operations regardless of their semantics. It starts by studying the impact of the most important optimization: the elimination of public-key cryptography. Then, it analyzes the impact of the optimizations described in Section 5.1.

Elimination of Public-Key Cryptography

To evaluate the benefit of using MACs instead of public key signatures, we implemented a version of the library that uses the BFT-PK algorithm. The version of BFT-PK described in Chapter 2 relies on the extra power of digital signatures to authenticate pre-prepare, prepare, checkpoint, and view-change messages but it can be modified easily to use MACs to authenticate other messages. Our implementation of BFT-PK is identical to the BFT library but it uses public-key signatures to authenticate these four types of messages. This allowed us to measure the impact of the more subtle part of this optimization.

The experiments compared the latency and throughput of two implementations of the simple service: the one labeled BFT used the BFT library and the one labeled BFT-PK used the BFT-PK library. We only compared performance of read-write operations because both libraries have the same performance with the read-only optimization.

Table 8.5 reports the latency to invoke an operation when the simple service is accessed by a single client. The results were obtained by timing a large number of invocations in three separate runs. We report the average of the three runs. The standard deviations were always below 0.5% of the reported value.

system	0/0	0/4	4/0
BFT-PK	59368	59761	59805
BFT	431	999	1046

Table 8.5: Cost of public-key cryptography: operation latency in microseconds.

BFT-PK has two signatures in the critical path and each of them takes 29.4 ms to compute. BFT eliminates the need for these signatures and achieves a speedup between 57 and 138 relative to BFT-PK. We use the SFS [MKKW99] implementation of a Rabin-Williams public-key cryptosystem with a 1024-bit modulus to sign messages and verify signatures. There are other public-key cryptosystems that generate signatures faster, e.g., elliptic curve public-key cryptosystems, but signature verification is slower [Wie98] and in our algorithm each signature is verified many times.

Theoretically, BFT-PK scales better than BFT as the number of replicas increases because the latency in BFT-PK grows linearly with the number of replicas rather than with the square of this number. But in practice BFT-PK only outperforms BFT for an unreasonably large number of replicas. For example, the performance model predicts that BFT's latency for operation 0/0 with 280 replicas is still lower than BFT-PK's latency with 4 replicas.

Figure 8-11 compares the throughput of the two implementations of the simple service for operations with different argument and result sizes. It uses the experimental setup and methodology described in Section 8.3.2: there are 5 client machines and 4 replicas. Each point in the graph is the average of at least three independent runs and the standard deviation for all points was below 4% of the reported value (except that it was as high as 17% for the last four points in the graph for BFT-PK operation 4/0).

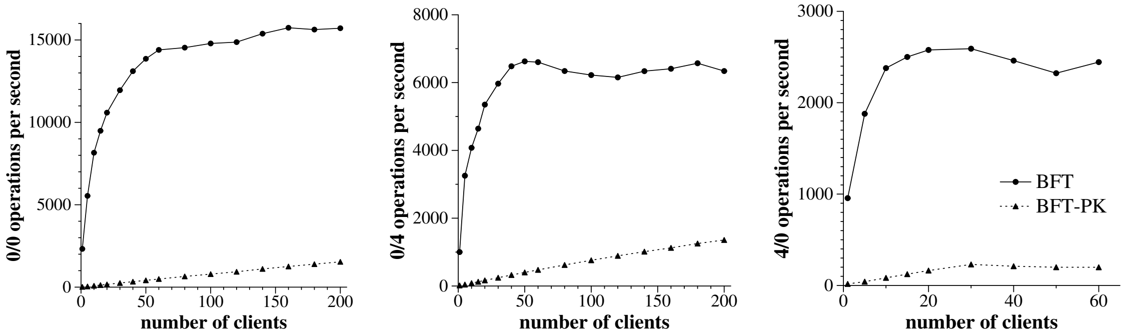

Figure 8-11: Cost of public-key cryptography: throughput in operations per second.

The throughput of both implementations increases with the number of concurrent clients because of request batching. Batching amortizes the signature generation overhead in BFT-PK over the size of the batch. Since this overhead is independent of the batch size, the throughput of the two implementations grows closer as the batch size increases. The current implementation limits batch size to 101 requests in operations 0/0 and 0/4 and 16 requests in operation 4/0; the throughput of both implementations saturates once the batch size reaches its maximum. The maximum throughput achieved by BFT-PK is 5 to 11 times worse than the one achieved by BFT.

If there were no limits on batch size, the two implementations would theoretically reach similar throughput values. However, this could only happen with an unreasonably large number of concurrent clients.

Digest Replies

To evaluate the impact of the digest replies optimization described in Section 5.1.1, we modified the BFT library not to use this optimization. This section compares the performance of two implementations of the simple service: BFT, which uses the regular BFT library, and BFT-NDR, which uses the version of the library without the digest replies optimization.

Figure 8-12 compares the latency to invoke the two implementations of the simple service as the size of the operation result increases. The standard deviations were always below 3% of the reported value. The digest replies optimization reduces the latency to invoke operations with large

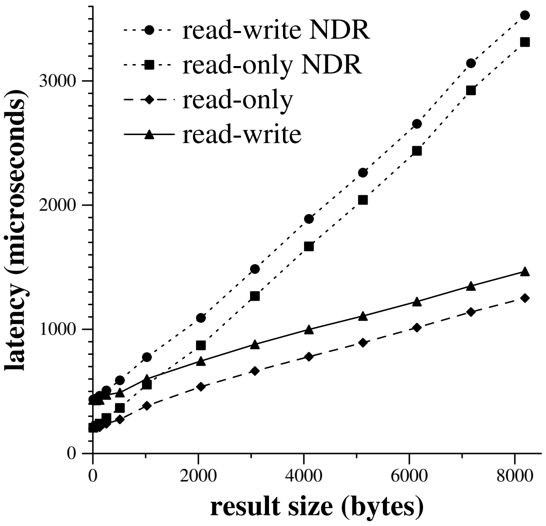

Figure 8-12: Latency with varying result sizes with and without the digest replies optimization. The lines labeled NDR correspond to the configuration without the optimization.

results significantly: it speeds up execution by up to a factor of 2.6.

The performance benefit of the digest replies optimization increases linearly with the number of replicas. In BFT-NDR, all replicas send back replies with the operation result to the client; whereas in BFT only one replica sends back a reply with the result and the others send small digests. Therefore, the speedup afforded by the optimization is approximately equal to $2f + 1$ with large result sizes.

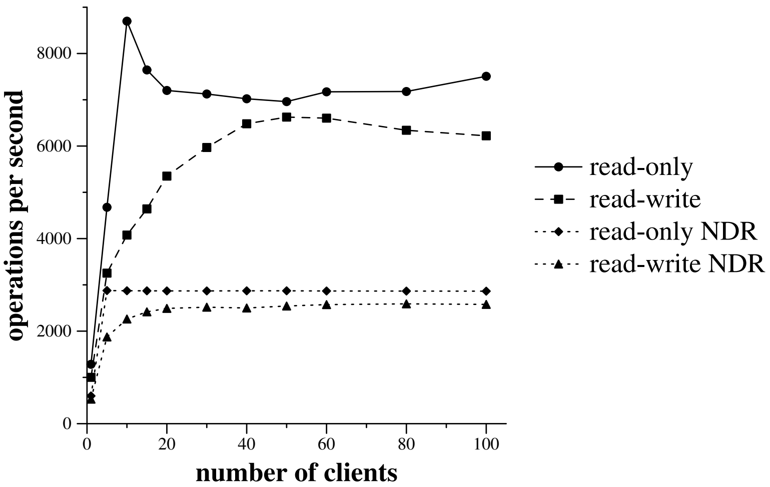

Figure 8-13: Throughput for operation 0/4 with and without the digest replies optimization. The lines labeled NDR correspond to the configuration without the optimization.

Figure 8-13 shows throughput results for operation 0/4. The values in the figure for BFT are the same that appeared in Figure 8-8. The standard deviation for the BFT-NDR values was always below 2% of the reported value.

BFT achieves a throughput up to 3 times better than BFT-NDR. The bottleneck for BFT-NDR is the link bandwidth: it is limited to a maximum of at most 3000 operations per-second regardless of the number of replicas. The digest replies optimization enables the available bandwidth for sending replies to the clients to scale linearly with the number of replicas and it also reduces load on replicas' CPUs.

Request Batching

The throughput results have shown the importance of batching requests and running a single instance of the protocol to prepare the batch. However, we did not present a direct comparison between the performance of the service with and without request batching; Figure 8-14 offers this comparison for the throughput of operation 0/0. Without batching, the throughput does not grow beyond 3848 operations per second and starts to decrease with more than 20 clients. The experiments in the previous section show that throughput reaches 15740 operations per second with batching.

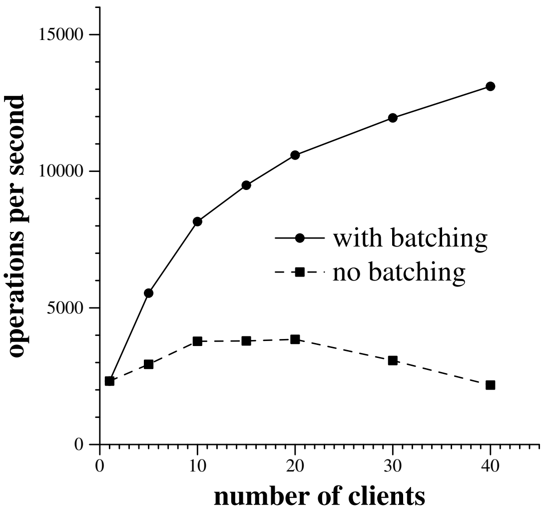

Figure 8-14: Throughput for operation 0/0 with and without request batching.

Since the replication algorithm can process many requests in parallel, the throughput without batching grows with the number of clients up to a maximum that is 66% better than the throughput with a single client. But processing each of these requests requires a full instance of the prepare protocol; and the replica's CPUs saturate for a small number of clients hindering throughput.

For our experimental environment, the best configuration uses a batching window of 1: the primary waits until the requests in a batch execute before sending a pre-prepare message for the next batch. In WAN environments where the latency is higher, the window should be set to a larger value to allow several batches to be processed in parallel.

Separate Request Transmission

The BFT library sends small requests inlined in pre-prepare messages but requests with argument size greater than 255 bytes are not inlined. These requests are multicast by the client to all replicas and the primary only includes their digests in pre-prepare messages. We measured the impact on latency and throughput of separating request transmission.

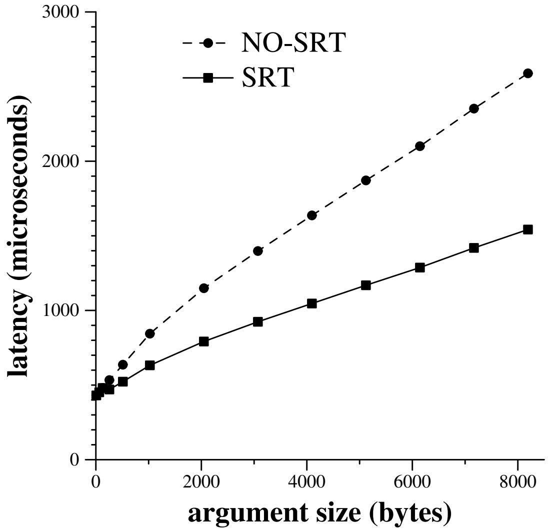

Figure 8-15: Latency for varying argument sizes with separate request transmission, SRT, and without, NO-SRT.

Figure 8-15 compares the latency to invoke the simple service for varying argument sizes with and without separate request transmission. Separating request transmission reduces latency by up to 40% because the request is sent only once and the primary and the backups compute the request's digest in parallel. The performance model predicts that the reduction will increase towards an asymptote of 53% as the argument size increases.

The other benefit of separate request transmission is improved throughput for large requests. Figure 8-16 compares the throughput for operation 4/0 with and without separate request transmission. It shows that the optimization improves throughput by up to 91%. This happens because the requests go over the network twice when they are inlined in pre-prepare messages: once from the client to the primary and then from the primary to the backups. Additionally, inlining the requests results in a maximum batch size of 2 (due to the limit on the size of pre-prepares).

Other Optimizations

The tentative execution optimization eliminates one round of the protocol: it allows replicas to execute requests and send replies to clients as soon as requests prepare. We implemented one version of the simple service, BFT-NTE, that uses the BFT library modified not to execute requests tentatively.

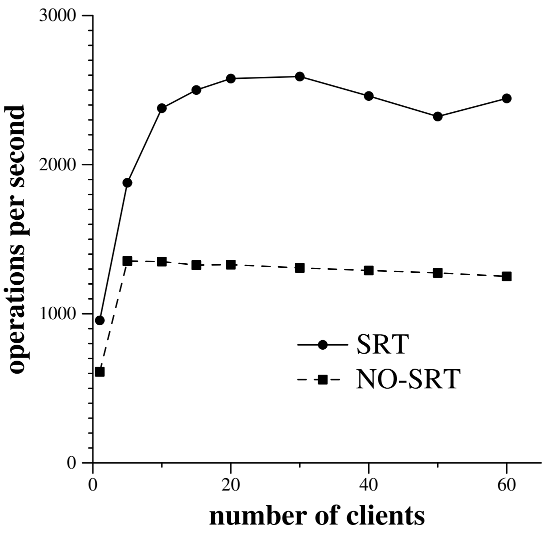

Figure 8-16: Throughput for operation 4/0 with separate request transmission, SRT, and without, NO-SRT.

We measured the latency of the BFT-NTE service as the argument and result sizes vary between 8 B and 8 KB. The tentative execution of requests reduces latency by a value that does not depend on the size of argument and result values. Therefore, the impact of this optimization decreases as the argument or result size increases. For example, the optimization improves performance by 27% with 8 B argument and result sizes but only by 5% when the argument size increases to 8 KB. We also measured the throughput of operations 0/0, 0/4, and 4/0 without tentative execution. The results show that this optimization has an insignificant impact on throughput.

We conclude that tentative execution of requests does not improve performance as significantly as the previous optimizations did (in our experimental setup). Even in WAN environments where communication latency is higher, this optimization should not improve service latency by more than 20% (because it eliminates one message delay from a total of 5). Since the throughput in these environments is also lower, the performance gain should be significantly smaller than this maximum.

A potential benefit of tentative execution of requests is that it enables the piggybacking of commit messages on pre-prepare and prepare messages. We implemented a version of the simple service with piggybacked commits and measured its latency and throughput. This optimization is not part of the BFT library; we only wrote code for it to work in the normal case.

Piggybacking commits has a negligible impact on latency because the commit phase of the protocol is performed in the background thanks to tentative execution of requests. It also has a small impact on throughput except when the number of concurrent clients accessing the service is small. For example, Figure 8-17 compares the throughput for operation 0/0 with and without this optimization. Piggybacking commits improves throughput by 33% with 5 clients and by 27%

with 10 but only by 3% with 200 clients. The benefit decreases with the number of clients because batching amortizes the cost of processing the commit messages over the batch size.

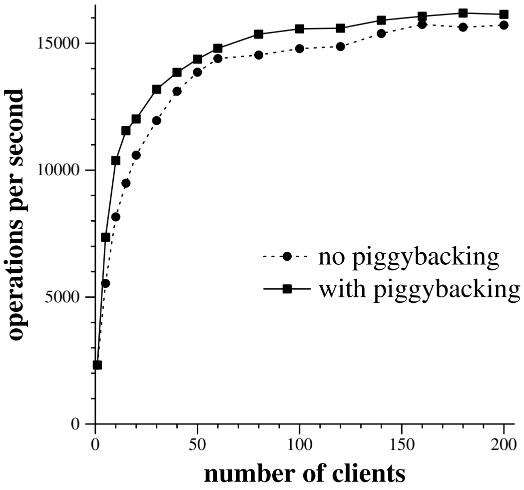

Figure 8-17: Throughput for operation 0/0 with and without piggybacked commits.

8.3.4 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. This section describes experiments to measure the latency and throughput of a system with seven replicas ($f = 2$) and uses the analytic performance model to predict performance with more replicas.

Latency

We ran experiments to measure the latency with varying argument and result sizes with 7 replicas and compared these results with the ones obtained with 4 replicas. In both configurations, all the replicas had a 600 MHz Pentium III processor and the client had a 700 MHz Pentium III processor.

Varying argument size. Figure 8-18 compares the latency to invoke the replicated service with $f = 1$ (4 replicas) and $f = 2$ (7 replicas) as the size of the operation argument increases while keeping the result size fixed at 8 bytes. The figure has two graphs: the first one shows elapsed times and the second shows the percentage slowdown of the configuration with $f = 2$ relative to the configuration with $f = 1$. The standard deviation was always below 2% of the reported value. It is not clear why the slowdown drops for argument sizes of 5 KB and 6 KB with the read-only optimization.

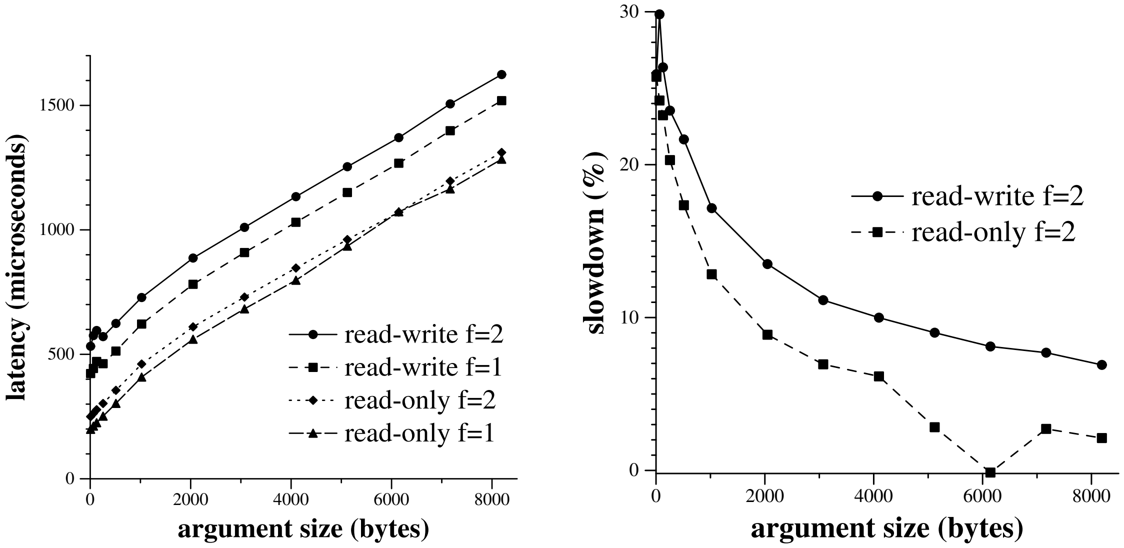

Figure 8-18: Latency with varying argument sizes with $f = 2$: absolute times and slowdown relative to $f = 1$.

The results show that the slowdown caused by increasing the number of replicas to 7 is low. The maximum slowdown for the read-write operation is 30% and it is 26% for the read-only operation. The results also show that the slowdown decreases as the argument size increases: with an argument size of 8 KB, the slowdown is only 7% for the read-write operation and 2% with the read-only optimization. According to the performance model, increasing the number of replicas introduces an overhead that is independent of the size of the operation argument; this explains why the slowdown decreases as the argument size increases.

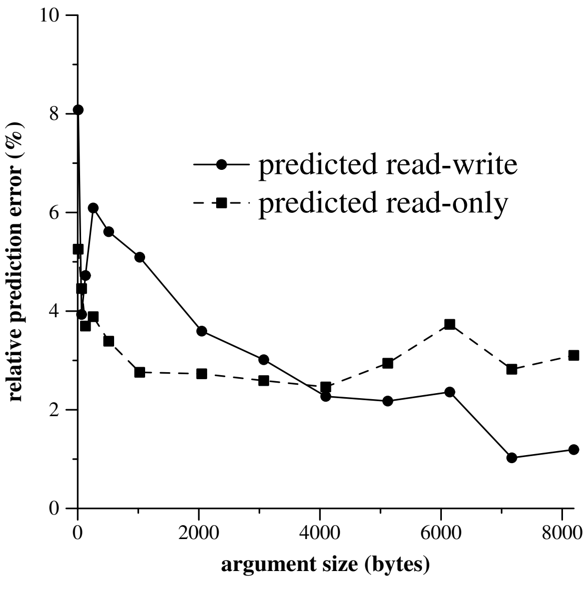

Figure 8-19: Latency model: relative prediction error for varying argument sizes with $f = 2$.

The latency model can predict these experimental results accurately. Figure 8-19 shows the error of the latency values predicted by the model for $f = 2$ relative to the values measured. The error is always below 8% and it is significantly lower for most argument sizes.

Since the model proved to be quite accurate, we used it to predict latency for configurations with more replicas. Figure 8-20 shows the predicted slowdown relative to the configuration with $f = 1$ for configurations with increasing values of f. The slowdown increases linearly with the number of replicas for read-only operations. For read-write operations, the slowdown increases with the square of the number of replicas but with a small constant. Since the overhead due to adding more replicas is independent of the argument size, the slowdown decreases as the argument size increases: for example, the slowdown for the read-write operation with $f = 10$ is 4.2 with 8 byte arguments, 2.3 with 4 KB, and only 1.9 with 8 KB.

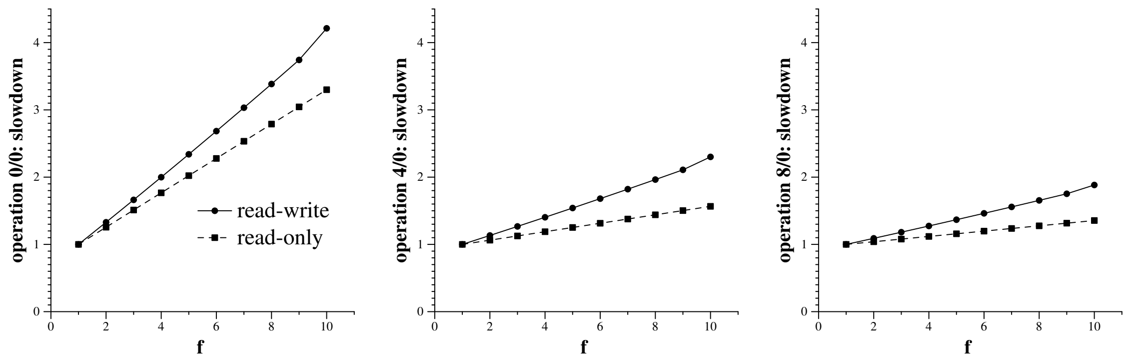

Figure 8-20: Predicted slowdown relative to the configuration with $f = 1$ for increasing f and argument size.

Varying result size. We also measured the latency for varying result sizes with $f = 2$; Figure 8-21 compares these results with those obtained with $f = 1$. The figure has two graphs: the first one shows elapsed times and the second shows the percentage slowdown of the configuration with $f = 2$ relative to the configuration with $f = 1$. The values are averages of 5 independent runs and the standard deviation was always below 2% of the reported averages.

Like in the case of varying argument sizes, the results show that the slowdown caused by increasing the number of replicas to 7 is small: the maximum slowdown for both read-only and read-write operations is 26%. The digest-replies optimization makes the overhead introduced by increasing the number of replicas independent of the result size. Therefore, the slowdown also decreases as the result size increases: the slowdown with 8 KB results is 5% for the read-write operation and only 1% with the read-only optimization.

The digest-replies optimization has another interesting effect: the communication time for the large reply with the result hides the time to process the small replies with the digests. Because of this effect, the slowdown drops faster as the result size increases than it does when the argument size increases. This effect is clear with the slowdown for the read-only operation.

Figure 8-22 shows that the performance model is less accurate at predicting the latency for

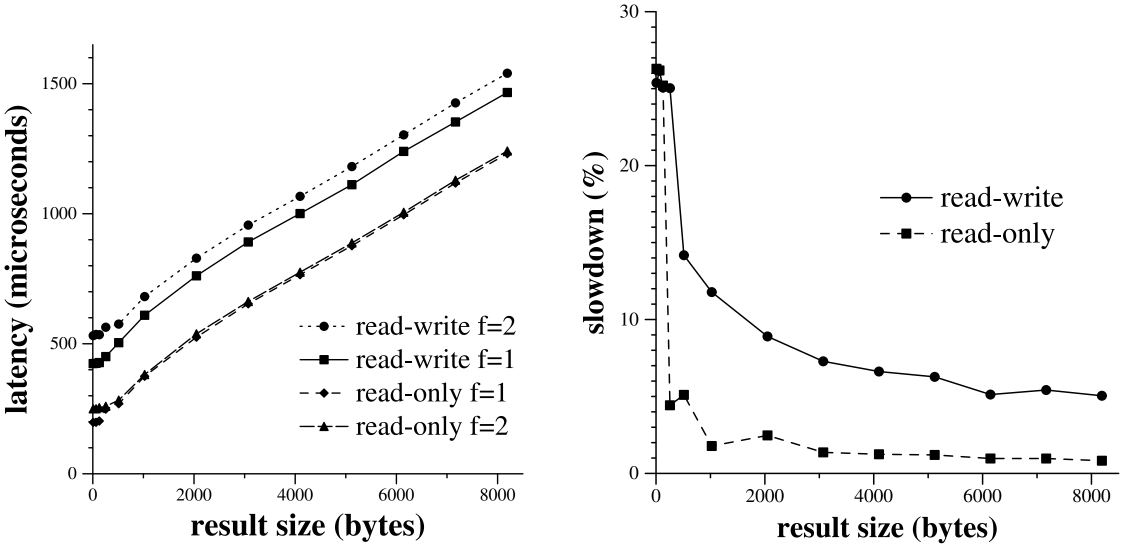

Figure 8-21: Latency with varying result sizes with $f = 2$: absolute times and slowdown relative to $f = 1$.

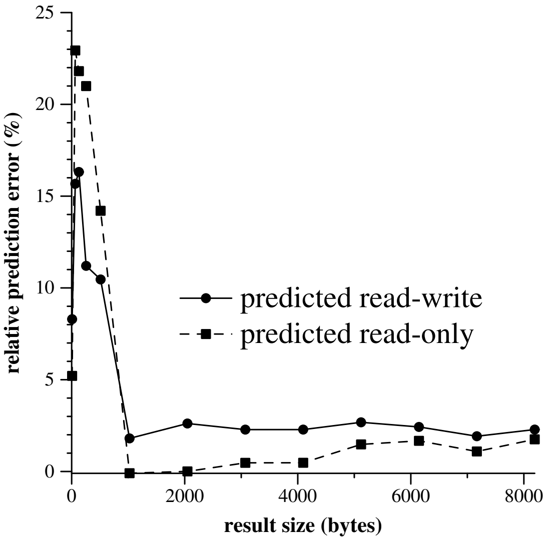

Figure 8-22: Latency model: relative prediction error for varying result sizes with $f = 2$.

$f = 2$ as the result size increases. The error is as high as 23% for small result sizes but it is less than 3% for result sizes greater than 512 bytes. This experimental configuration uses a client that is faster than the machines where the parameters for the model were measured; this can explain the large error for small result sizes (for larger result sizes this error is hidden because the cost of processing digest replies is overlapped with the communication time for the reply with the result).

The performance model is sufficiently accurate to make interesting predictions for configurations with more replicas. Figure 8-23 shows the predicted slowdown relative to the configuration with $f = 1$ for operations 0/0, 0/4, and 0/8. The results for operation 0/4 and 0/8 are similar to those presented for operations 4/0 and 8/0. The difference is that the slowdown grows slower as the number of replicas increases. This happens because the time to process the small replies is hidden by the communication time for the reply with the result for large result sizes.

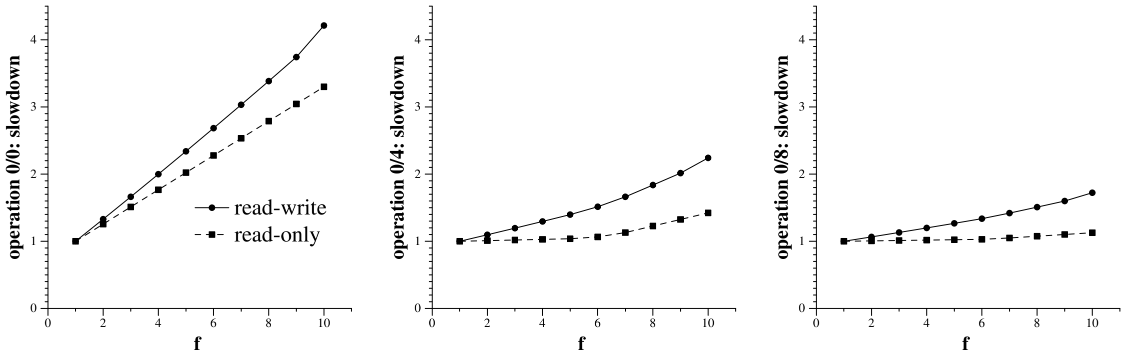

Figure 8-23: Predicted slowdown relative to the configuration with $f = 1$ for increasing f and result size.

Throughput

We tried to measure the throughput of the system configured with $f = 2$. But since this configuration requires 7 replicas, the experiments were limited to use 2 machines to run the processes that simulate the client population. This prevented us from obtaining meaningful results because the CPU of the client machines and their links to the switch became bottlenecks.

The performance model was able to predict the maximum throughput for $f = 1$ and the latency for $f = 2$ with good accuracy. Therefore, we are confident that it provides a good prediction for the maximum throughput in configurations with more replicas; Figure 8-24 shows this prediction for operations 0/0, 0/4, and 4/0. The prediction was obtained for 100 client machines with a batch size of 100 for operations 0/0 and 0/4, and with a batch size of 16 for operation 4/0.

The figure suggests that increasing the value of f up to 10 does not cause a severe throughput degradation. To explain this, it is necessary to look at the components of the model in more detail. The model breaks the time to execute the requests into three components: the time to get the requests in the batch to the replicas, T^b_{req}, the time to prepare the batch, T^b_{prep}, and the time to execute the

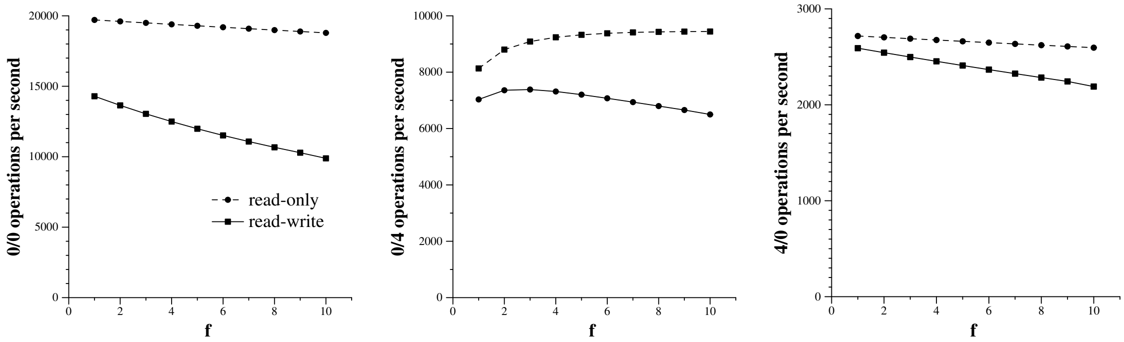

Figure 8-24: Predicted throughput for increasing f for operations 0/0, 0/4 and 4/0.

requests in the batch and get the replies to the clients T^b_{erep}.

For our experimental setup and the values in this figure, the last component is equal to the CPU time spent by the replicas executing the requests and sending the replies. Therefore, T^b_{erep} does not increase with the number of replicas. T^b_{req} is either equal to the communication time in each replica's link (in operation 4/0) or to the CPU time receiving and checking the requests at the replicas (in operations 0/0 and 0/4). In either case, T^b_{req} grows slowly with the number of replicas; it grows only because of increased communication cost due to larger authenticators. T^b_{prep} grows quickly as the number of replicas increases because both the number and size of pre-prepare/prepare messages processed by the replicas grow linearly with f. But the growing overhead in T^b_{prep} is amortized over the size of the batch.

The T^b_{prep} component is 0 for read-only requests, which explains why the throughput decreases more slowly with the read-only optimization for operations 0/0 and 4/0. Additionally, T^b_{erep} actually decreases with the number of replicas for operation 0/4, which explains why throughput improves slightly as the number of replicas increases.

For read-write operations 0/0 and 0/4, the current implementation might not do as well as the model predicts because the requests in these operations are inlined in the pre-prepare message and the maximum batch size would decrease down to 27 for $f = 10$. But this is not an intrinsic problem; the library could use separate request transmission for all request sizes.

8.3.5 Sensitivity to Variations in Model Parameters

We used the analytic model to predict the performance of the BFT library in two different experimental setups: a WAN environment, and a LAN with 1Gb/s Ethernet and 1.2GHz processors. The WAN environment is interesting because placing the replicas in different geographic locations is an important technique to increase their failure independence. The LAN environment represents the fastest LAN available today.

WAN

We assumed that the only parameters that varied when switching between our current experimental setup and the WAN environment were the network latency, S_f, and the network cost per byte, S_v, (i.e., the inverse of the throughput). We also assumed that these parameters were the same for communication between all pairs of nodes.

We measured the value of these parameters between a host at MIT and a host at the University of California at Berkeley. We obtained a round-trip latency of 75ms and a throughput of approximately 150KB/s. Based on these values, we set $S_f = 37500\mu$s and $S_v = 6.61\mu$s/byte.

We are not modeling message losses. We measured a loss rate of less than 0.5%; this should not impact performance very significantly. Furthermore, the algorithm can tolerate some message loss without requiring retransmissions. We are also assuming that multicast works in the WAN environment; this is not true in the entire Internet today but there are already several important ISPs that provide multicast services (e.g. UUNET).

Figure 8-25 shows the predicted slowdown in the latency to invoke the replicated service, BFT, relative to the service without replication, NO-REP, in a WAN. It presents results for operations 0/0, 0/8, and 8/0 with and without the read-only optimization. The number of replicas was four.

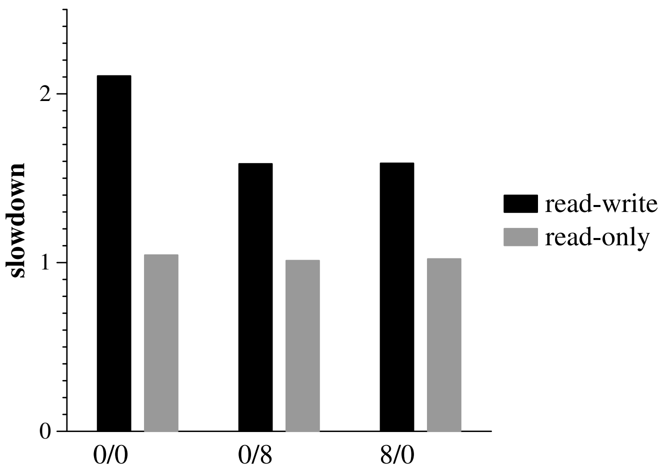

Figure 8-25: Latency: predicted slowdown due to BFT library in a WAN environment.

In the LAN, we measured a slowdown of approximately 4 for operation 0/0 without the read-only optimization and 2 with the optimization. The slowdown decreases in the WAN because the CPU costs are dwarfed by the network costs. The slowdown is approximately 2 for read-write operation 0/0 because the protocol introduces an extra round-trip delay relative to the system without replication. The read-only optimization eliminates the extra round-trip and virtually eliminates the slowdown.

The slowdown for read-write operations 0/8 and 8/0 is actually slightly larger than the value we measured in our experimental setup. This is because the ratio between a round-trip delay and the time to transmit an 8 KB message is higher in the WAN environment. However, the slowdown in the WAN should virtually vanish for larger result and argument sizes whereas it tends to an asymptote of 1.26 in our LAN. In many configurations, communication between the replicas is likely to be faster than communication between clients and replicas. This would decrease slowdown even further.

The throughput in the WAN environment is bound by the low network throughput in our model. The extra round-trip latency introduced by the protocol is amortized over the batch size and we can run the protocol in parallel for several batches. Thus, the limit is the network throughput in the server links not the extra computation and communication introduced by the protocol. For example, the server link bandwidth limits the throughput in NO-REP to 18 operations per second in operation 0/8. The predicted throughput for BFT is 59 operations per second without the read-only optimization and 65 operations per second with the optimization.

Fast LAN

To model the LAN with 1Gb/s Ethernet and 1.2GHz processors, we divided the switch parameters we measured by 10 and the processor parameters by 2. Figure 8-26 shows the predicted slowdown in the latency to invoke the replicated service, BFT, relative to the service without replication, NO-REP, in the fast LAN environment. It presents results for operations 0/0, 0/8, and 8/0 with and without the read-only optimization. The number of replicas was four.

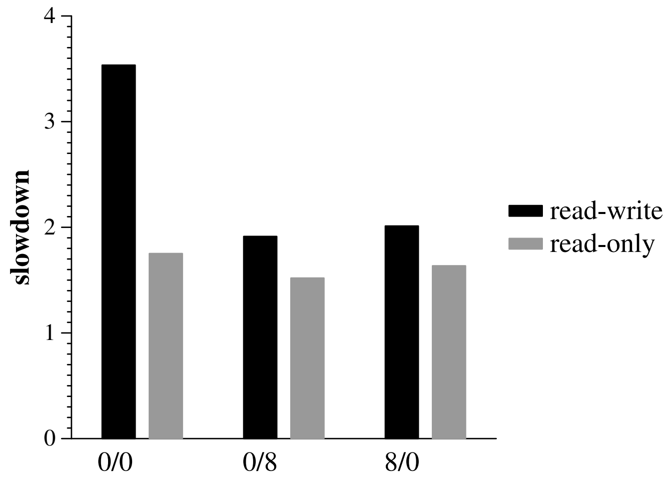

Figure 8-26: Latency: predicted slowdown due to BFT library in a fast LAN environment.

The predictions for the slowdown in operation 0/0 in the fast LAN environment are almost identical to those in our experimental environment. But the slowdown for operations 0/8 and 8/0 is higher. This is explained by a higher ratio between the cost per byte of digest computation and the cost per byte of communication. The model predicts an asymptote of 1.65 for the slowdown as the argument and result sizes increase whereas it predicts an asymptote of 1.26 in our experimental environment.

Figure 8-27 shows the predicted throughput for BFT in our experimental environment and in the fast LAN. The throughput is normalized to allow a comparison: it is divided by the predicted throughput for NO-REP in the same configuration.

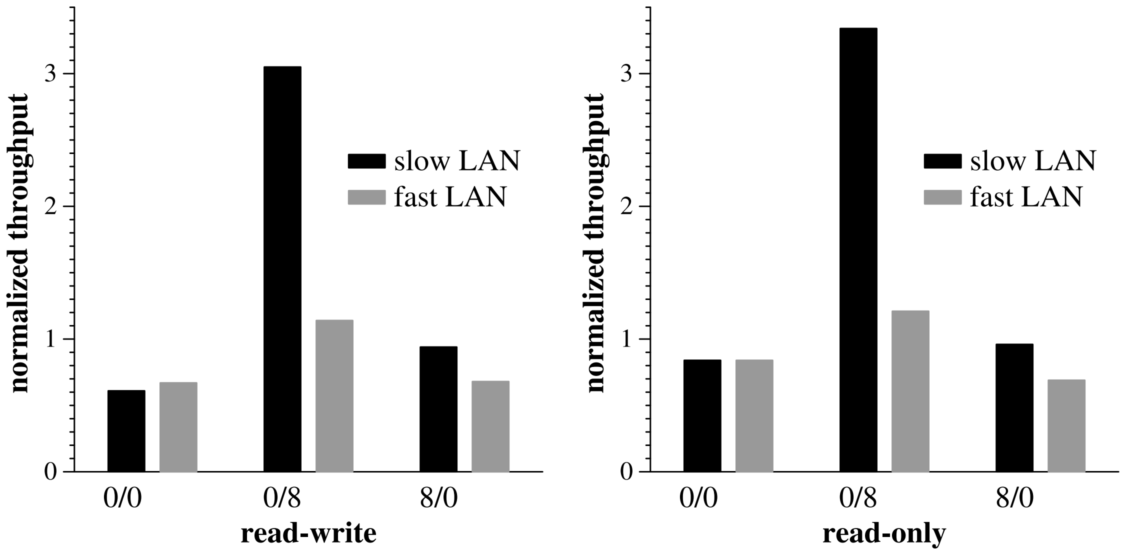

Figure 8-27: Predicted throughput for BFT in slow and fast LANs normalized to NO-REP's throughput.

The normalized throughputs for operation 0/0 in the two configurations are very similar because the server CPU is the bottleneck for both BFT and NO-REP in the two configurations. But the normalized throughput for operations 0/8 and 8/0 is lower in the fast LAN. This happens because the network speed increases by a factor of 10 but the CPU speed only increases by a factor of 2 and BFT places a heavier load on the CPUs than NO-REP.

8.4 Checkpoint Management

The experiments in the previous section 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 large memory-mapped file. The service operations can read or write these pages.

The 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.

8.4.1 Checkpoint Creation

The BFT library creates a checkpoint whenever the requests in a batch with sequence number divisible by the checkpoint period are executed. The requests that execute between two checkpoints are said to be in the same *checkpoint epoch*. The checkpoints are created using the technique described in Section 5.3. In our experimental setup, the checkpoint period, K, is equal to 128. The state partition tree has 4 levels, each internal node has 256 children, and the pages (i.e. the leaves of the tree) have 4 KB.

We ran a benchmark to measure the cost of checkpoint creation using the simple service with state. The benchmark used a state with 256 MB, 4 replicas, and 1 client. The client invoked operations that received an offset into the state and a stride as arguments; and then wrote eight 4-byte words to the state starting at the offset and separated by the stride. The offset argument for an operation was made equal to the offset of the last word written by the previous operation plus the stride value. This allowed us to measure the cost of checkpointing in a controlled way: by running experiments with different stride values, we were able to vary the number of modified pages per checkpoint epoch without changing the cost to run the protocol and execute the operations.

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 8-28 shows the values we measured for these times with a varying number of modified pages per checkpoint epoch. The time to create checkpoints increases slightly when the modified pages are selected at random (for example, it increases 4% for 128 pages).

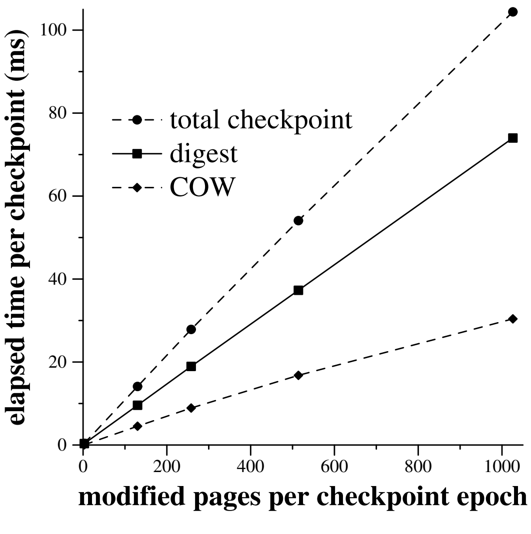

Figure 8-28: Checkpoint cost with a varying number of modified pages per checkpoint epoch.

The results show that both the time to perform copy-on-write and the time to compute digests grow linearly with the number m of distinct pages modified during a checkpoint epoch. We ran a linear regression on the digest and copy-on-write results. The coefficient of determination was 1 for the digest results and 0.996 for the copy-on-write results. We obtained the following model for the checkpoint time in microseconds:

$T_{chcpt}(m) = T_{digest}(m) + T_{cow}(m)$

$T_{digest}(m) = 248 + 72 \times m$

$T_{cow}(m) = 767 + 29 \times m$

T_{digest} includes the time to iterate over a bitmap that indicates which pages have been modified and the time to clear this bitmap; this accounts for the 248μs latency. The cost to digest each page is 72μs, which is 39% higher than the time to digest a page using MD5. The additional overhead is due to the cost of updating the incremental checkpoint for the parent using the AdHash [BM97] algorithm.

T_{cow} includes the time to allocate memory to hold a copy of the page and the time to copy the page. The model for T_{cow} is not as good because the cost per page actually increases with the number of pages modified; this accounts for the high latency of 767μs in spite of an experimental result of 52μs with $m = 3$. We ran some micro-benchmarks that showed that the increased cost per page was due to a growing cost to allocate memory to hold the copy of the page.

In these experiments, the service state fit in main memory. We do not expect checkpointing to increase the number of disk accesses significantly when the state does not fit in main memory. A page is copied just before it is accessed and digests are computed on the pages that have been modified in the preceding checkpoint epoch; these pages are likely to be in main memory. The only case where checkpointing can increase the number of disk accesses significantly is when the space overhead to keep the checkpoints represents a significant fraction of the memory available; this case is unlikely in practice.

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. For example, the cost of checkpoint creation represents approximately 65% of the total cost to run the experiment with a stride of 1024. This is a worst-case example because each operation modifies 8 pages without performing any computation and with little communication overhead (because it has small argument and result sizes). Nevertheless, it is not hard to imagine real applications where the current implementation of checkpoint management will be the bottleneck.

It is possible to improve checkpoint performance with sparse writes by using smaller pages in the partition hierarchy. But decreasing the size of these pages increases the space overhead due to additional meta-data. A more interesting alternative would be to compute checkpoint digests lazily. It is possible to modify the protocol not to send checkpoint digests in checkpoint messages. Thus, checkpoint digests would need to be computed only before a view change or a 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.

8.4.2 State Transfer

We also ran experiments to measure the time to complete a state transfer. The experiments used the simple service with 256 MB of state and 4 replicas. In the first experiment, 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 pages chosen sequentially. Figure 8-29 shows the elapsed time to complete the state transfer and its throughput.

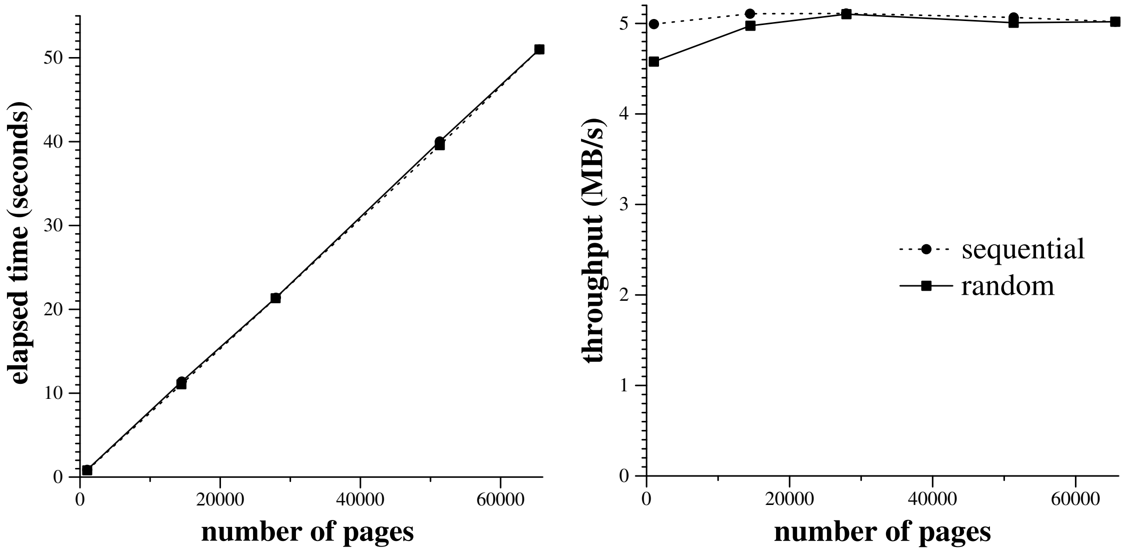

Figure 8-29: State transfer latency and throughput.

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 1000 random pages. The throughput is lower with random pages because it is necessary to fetch more meta-data 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 their correctness. We measured an average time to digest each page of $56\mu s$ and our communication model predicts $651\mu s$ to send the fetch message and receive the data. This predicts a throughput of 5.5 MB/s, which is close to the maximum throughput observed (5.1MB/s).

The second experiment ran 5 clients. Each client invoked an operation that took a 4 KB page as an argument and wrote its value to a random page in the state. We ran this experiment with 3 replicas and measured an aggregate throughput of 6.7 MB/s from the clients to the service. Then, we reran the experiment with 4 replicas but one of the replicas was started 25 seconds after the beginning of the experiment. The results show that the replica was unable to get up-to-date; it started a state transfer that never ended because the state was modified faster than it could fetch the modifications. This happened because the maximum state transfer throughout is approximately 5 MB/s and the current implementation does not give priority to fetch messages (it uses a single queue for all messages). On the positive side, the state transfer did not delay request processing significantly and the clients achieved an aggregate throughput of 6.5 MB/s.

The problem in the previous paragraph 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. First, the throughput of the state transfer mechanism can be improved by fetching pages in parallel from all replicas; this should improve throughput to the link bandwidth (12MB/s). Second, the replicas can give priority to handling of fetch requests: this will reduce the degradation in the state transfer throughput in the presence of request processing. Additionally, it will slow down request processing thereby increasing the chances that the replica will be able to complete the state transfer. A more drastic step would be to artificially restrict the rate of change.

8.5 View Changes

The experiments described so far analyze the performance of the system when there are no failures. 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 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 and 4 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 took a 4KB page value as an argument and wrote it 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 8.6 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.

	idle	10%	50%
view-change time (μs)	575	4162	7005

Table 8.6: Average view change time with varying write percentage.

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 small; it is only 34% greater than the time to execute operation 0/0 on the simple service.

The view change time increases when the replicas process client requests because view-change messages include information about requests that are prepared or pre-prepared by the replicas. Table 8.7 shows that the average size of view changes increases: they contain information about an average of 56 requests for 10% writes and 71 requests for 50% writes. The increase in the view change time from 10% to 50% writes is partly explained by the 27% increase in the number of requests in view change messages but most of it is due to one view change that took 607ms to complete. This view change was much slower 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 time to complete view changes also increases when it is necessary to fetch missing requests or when the replica has to rollback its state because it executed a request tentatively that did not commit. But these are relatively uncommon occurrences.

	idle	10%	50%
view-change size (bytes)	160	1954	2418
new-view size (bytes)	136	189	203

Table 8.7: Average size of view-change and new-view messages with varying write percentage.

The time to complete a view change when the primary fails has an additional component: the timeout replicas wait for an outstanding request to execute before suspecting that the primary is faulty. The cost of the view change protocol in our library is small; this enables the timeout to be set to a small value (e.g., one second or less) to improve availability without risking poor performance due to false failure suspicions.

8.6 BFS

We measured the performance of the BFT library using simple, service-independent benchmarks. Next, we present the results of a set of experiments to evaluate the performance of a real service — BFS, which is a Byzantine-fault-tolerant NFS service built using the BFT library that was described in Section 6.3.

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 meta-data before replying to the client (as required by the NFS protocol [S^+85]), 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.

8.6.1 Experimental Setup

The experiments to evaluate BFS used the setup described in Section 8.1. They ran two well-known file system benchmarks: the modified Andrew benchmark [Ous90, HKM$^+$88] and PostMark [Kat97].

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

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

PostMark [Kat97] 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 sub-transactions: 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 sub-transaction are selected randomly with uniform probability distribution. The create operation creates a file with a random size within the configurable range. The delete operation deletes a random file from the pool. The read operation reads a random file in its entirety. The append operation opens a random file, seeks to its end, and appends a random amount of data. After completing all the transactions, the remaining files are deleted.

We configured PostMark with an initial pool of 10000 files with sizes between 512 bytes and 16 Kbytes. The files were uniformly distributed over 130 directories. The benchmark ran 100000 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, 4096-

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 (see Section 6.3).

Out of the 18 operations in the NFS V2 protocol only `getattr` is read-only because the time-last-accessed attribute of files and directories is set by operations that would otherwise be read-only, e.g., `read` and `lookup`. We modified BFS not to maintain the time-last-accessed attribute in order to apply the read-only optimization to `read` and `lookup` operations. This modification violates strict Unix file system semantics but is unlikely to have adverse effects in practice.

8.6.2 Performance Without Recovery

We will now analyze the performance of BFS without view-changes or proactive recovery. We will start by presenting results of experiments that ran with four replicas and later we will present results obtained with seven replicas. We also evaluate the impact of the most important optimization in BFT, the elimination of public-key cryptography, on the performance of BFS.

Four Replicas

Figures 8-30 and 8-31 present results for Andrew100 and Andrew500, respectively, in a configuration with four replicas and one client machine. We report the mean of 3 runs of the benchmark. The standard deviation was always below 1% of the reported averages except for phase 1 where it was as high as 33%.

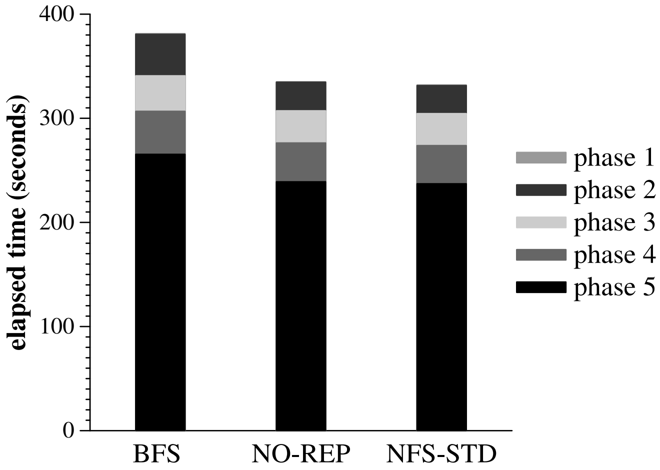

Figure 8-30: Andrew100: elapsed time in seconds.

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 what was observed with the latency of the simple service because the client spends a significant fraction of the elapsed time computing between operations (i.e., between receiving the reply to an operation and issuing the next request) and operations at the server perform some computation. Additionally, 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 Table 8.8 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 meta-data before replying to the client as required by the NFS protocol, whereas BFS ensures stability through replication.

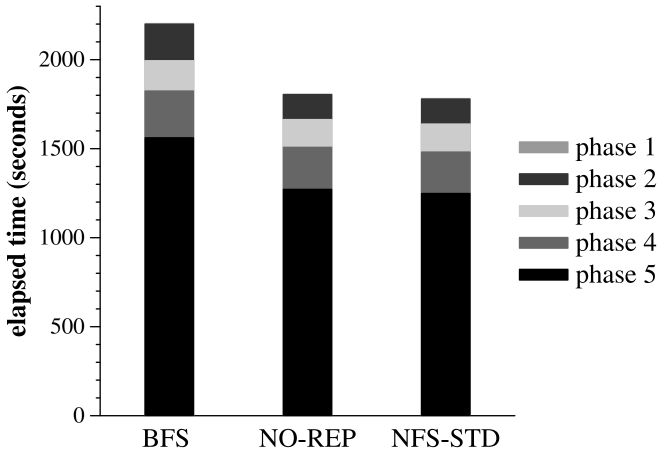

Figure 8-31: Andrew500: elapsed time in seconds.

Table 8.8 shows a comparison between BFS, NO-REP, and the NFS V2 implementation in Digital Unix, NFS-DEC. These experiments ran the Andrew benchmark with one client and four replicas on DEC 3000/400 Alpha workstations connected by a switched 10Mb/s Ethernet. The complete experimental setup is described in [CL99c].

The results show that BFS is 2% faster than NFS-DEC. This is because during phases 1, 2, and 5 a large fraction (between 21% and 40%) of the operations issued by the client are *synchronous*, i.e., operations that require the NFS implementation to ensure stability of modified file system state before replying to the client. NFS-DEC achieves stability by writing modified state to disk whereas BFS achieves stability with lower latency using replication (as in Harp [LGG+91]). NFS-DEC is faster than BFS in phases 3 and 4 because the client does not issue synchronous operations.

Figure 8-32 presents the throughput measured using PostMark. The results are averages of three runs and the standard deviation was below 2% 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

phase	BFS	NO-REP	NFS-DEC
1	0.47	0.35	1.75
2	7.91	5.08	9.46
3	6.45	6.11	5.36
4	7.87	7.41	6.60
5	38.3	32.12	39.35
total	61.07	51.07	62.52

Table 8.8: Andrew: BFS vs NFS-DEC elapsed times in seconds.

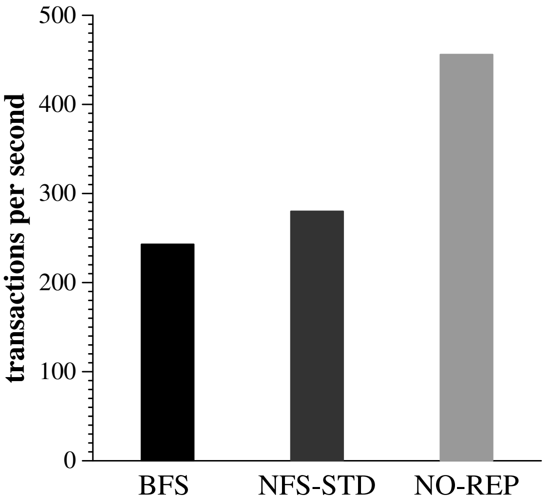

Figure 8-32: PostMark: throughput in transactions per second.

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.

Seven Replicas

Figure 8-33 shows a comparison between the time to complete Andrew100 with four replicas ($f = 1$) and with seven replicas ($f = 2$). All replicas had a 600 MHz Pentium III processor and the client had a 700 MHz Pentium III processor. We report the average of three runs of the benchmark. The standard deviation was always below 1% of the reported value.

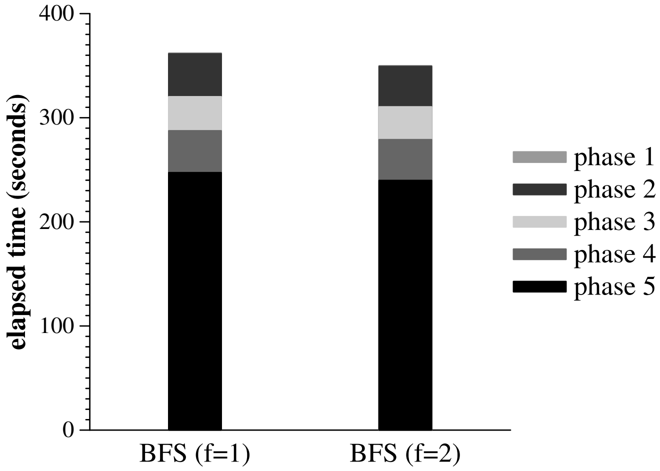

Figure 8-33: Andrew100: elapsed time with f=1 and f=2.

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. This outcome was predictable given the micro-benchmark results in the previous sections. Since there is a significant amount of computation at the client in Andrew100, BFS with $f = 2$ is only 3% slower than with $f = 1$.

Elimination of Public-Key Cryptography

The micro-benchmarks in Section 8.3.3 showed that the replacement of digital signatures by MACs improved performance dramatically. To evaluate the impact of this optimization on the performance of a real service, we implemented BFS-PK using the BFT-PK library (that was described in that section). Tables 8.9 and 8.10 present results comparing the time to complete Andrew100 and Andrew500 (respectively) in BFS and BFS-PK.

The results show that BFS-PK takes 12 times longer than BFS to run Andrew100 and 15 times longer to run Andrew500. The slowdown is smaller than the one observed with the micro-benchmarks because the client performs a significant amount of computation in this benchmark. Additionally, both BFS and BFS-PK use the read-only optimization for `getattr`, `read` and

phase	BFS-PK	BFS
1	25.4	0.7
2	1528.6	39.8
3	80.1	34.1
4	87.5	41.3
5	2935.1	265.4
total	4656.7	381.3

Table 8.9: Andrew100: elapsed time in seconds for BFS and BFS-PK.

`lookup`; this reduces the performance difference between BFS and BFS-PK during phases 3 and 4 where most operations are read-only.

phase	BFS-PK	BFS
1	122.0	4.2
2	8080.4	204.5
3	387.5	170.2
4	496.0	262.8
5	23201.3	1561.2
total	32287.2	2202.9

Table 8.10: Andrew500: elapsed time in seconds for BFS and BFS-PK.

8.6.3 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 co-processor, the read-only memory, and the watchdog timer in software. We are also simulating fast reboots. The LinuxBIOS project [Min00] 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`) [Min00]. 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 the recovery time for Andrew100 and Andrew500 with 30s reboots and with the period between key changes, T_k, set to 15s.

Table 8.11 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 8.11 presents a single line for both of them. The line labeled *restore state* only accounts for reading the log from disk; the service state pages are read from disk on demand when they are checked.

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

Table 8.11: Andrew: maximum recovery time in seconds.

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

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

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

Recovery Overhead

We also evaluated the impact of recovery on performance in the experimental setup described in the previous section; Figure 8-34 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.

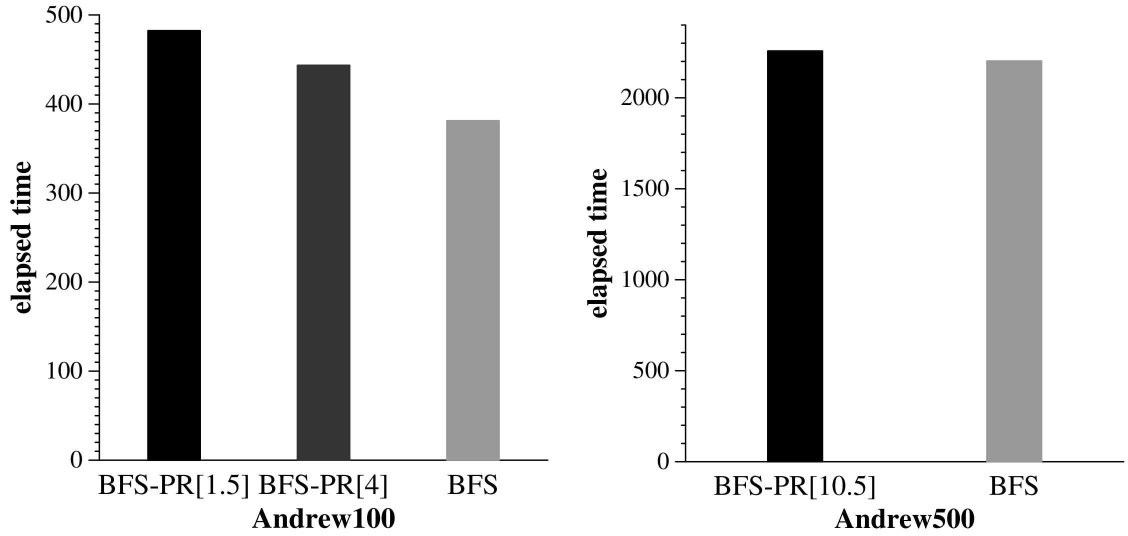

Figure 8-34: Andrew: elapsed time in seconds with and without proactive recoveries.

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

The results also show that the period between key changes, T_k, can be small without impacting performance significantly. T_k could be smaller than 15s but it should be substantially larger than 3 message delays under normal load conditions to provide liveness. The problem is that changing keys frequently does not scale well with the number of clients. Active clients need to refresh their keys to ensure that the reply certificates contain only messages generated by the replicas within an interval of size at most $2 \times T_k$. This means that, with 200 active clients and $T_k = 15s$, each replica would spend 20% of the time processing new-key messages from clients. This processing is performed by the secure co-processor, which allows the replicas to use the CPUs to execute the protocol. Nevertheless, it may be a problem with a large number of active clients.

8.7 Summary

The results in this chapter show that services implemented with the BFT library perform well even when compared with unreplicated implementations. Section 8.7.1 summarizes the experimental results obtained with the micro-benchmarks, which were designed to evaluate the performance of the BFT library in a service-independent way, and the performance results for BFS are summarized in Section 8.7.2.

8.7.1 Micro-Benchmarks

Recall that the micro-benchmarks compare two implementations of a simple service with no state and whose operations perform no computation. The two implementations are BFT, which is replicated using the BFT library, and NO-REP, which is not replicated. The micro-benchmarks overestimate the overhead introduced by the BFT library because, in real services, computation or I/O at clients and servers reduces the overhead relative to unreplicated implementations.

The experimental results show that our analytic performance model is accurate: the absolute value of the relative prediction error for latency and throughput was below 10% of the experimental results for almost all experiments.

Latency

When the operation argument and result sizes are very small, the latency to invoke the replicated service is much higher than without replication. The maximum slowdown relative to NO-REP occurs when the operation argument and result size are both equal to 8 B and it is equal to 4.07 for read-write operations and 1.93 with the read-only optimization.

However, the slowdown decreases quickly as the argument and result sizes increase. For example, the slowdown with an 8 KB result size is 1.47 for read-write operations and 1.25 with the read-only optimization. The model predicts an asymptote of 1.26 for the slowdown with very large arguments or results for both read-write and read-only operations. The read-only optimization reduces slowdown significantly with small argument and result sizes but its benefit decreases to zero as these sizes increase.

The model predicts similar trends in a WAN environment. However, the maximum predicted slowdown relative to NO-REP is approximately 2 for read-write operations because the communication latency in the WAN dwarfs CPU costs and BFT only adds an extra round-trip. The read-only optimization removes this round-trip and virtually eliminates the overhead.

Increasing the number of replicas from 4 to 7 does not cause a severe increase in the latency to invoke the replicated service. In our experimental setup, the maximum overhead relative to the configuration with 4 replicas is 30% with very small argument and result sizes. Furthermore, the overhead decreases as the argument or result sizes increase: it is at most 7% for read-write operations and 2% with the read-only optimization with an argument or result size of 8 KB.

The model predicts a similar behavior in configurations with up to 31 replicas ($f = 10$): there is a large overhead for operations with small argument and result sizes but it decreases as these sizes increase. For example, BFT with $f = 10$ is 4.2 times slower than with $f = 1$ with 8 B arguments and results but only 1.9 with 8 KB arguments and 1.7 with 8 KB results. The slowdown is lower with the read-only optimization: BFT with $f = 10$ is at most 3.3 times slower with 8 B arguments and results but only 1.35 with 8 KB arguments and 1.13 with 8 KB results.

Throughput

The results show that BFT has significantly lower throughput than NO-REP for operations with small argument and result sizes. The bottleneck in this case is the replica (or server) CPU and BFT generates more CPU load than NO-REP. For example, when both the argument and the result size are equal to 8 B, BFT achieves a throughput that is 52% lower than NO-REP's with read-write operations and 35% lower with the read-only optimization.

However, the throughput degradation is less significant with large argument sizes: BFT's throughput is only 11% lower than NO-REP's with 4 KB arguments. The bottleneck in this case is the network link to each replica (or to the server). Furthermore, with large result sizes BFT achieves better throughput than NO-REP because different clients can obtain large results in parallel from different replicas: BFT's throughput with 4 KB results is 2.3 times higher than NO-REP's for read-write operations and up to 3 times higher with the read-only optimization. The bottleneck in NO-REP is the network link to the client and in BFT it is the CPU at the replicas.

According to our model, increasing the resilience of the system to $f = 10$ does not cause a severe throughput degradation relative to the configuration with $f = 1$: the maximum degradation is 31% for read-write operations with very small argument and result sizes. Furthermore, it decreases as these sizes increase. The degradation is even lower for read-only operations: the maximum degradation is 5% and throughput actually improves as the number of replicas increases for large result sizes.

Impact Of Optimizations

BFT performs well because of several important optimizations. The most important is the elimination of public-key cryptography. This optimization improves latency dramatically in our experimental setup; it achieves a speedup between 57 and 138 depending on argument and result sizes.

The optimization also increases throughput by a factor of 5 to 11.

Batching requests, digest replies, and separate request transmission are also very important optimizations. Batching requests is very effective at improving throughput of read-write operations. For example, it improves the throughput of an operation with argument and result size equal to 8 B by a factor of 4.1. The digest replies optimization has a significant impact with large result sizes. Our results show that it reduces latency by up to a factor of 2.6 and improves throughput by up to a factor of 3. Similarly, separate request transmission improves latency and throughput significantly for operations with large argument sizes: it reduces latency by 40% and improves throughput by 91%.

Tentative execution of requests is not as effective: it improves latency by at most 27% and has no significant impact on throughput.

8.7.2 BFS

The performance results for BFS show that the relative overhead introduced by the BFT library is even lower for a real service. BFS takes 14% to 22% more time than an unreplicated service (which uses the same file system code) to complete scaled up versions of the Andrew benchmark.

The comparison with NFS implementations in production operating systems (Linux and Digital Unix) shows that BFS can be used in practice: its performs similarly to these systems, which are used daily by many users. BFS' performance ranges from 2% faster to 21% slower depending on the NFS implementation and the amount of data used in the scaled up Andrew benchmark. BFS is 2% faster than the NFS implementation in Digital Unix, which implements correct NFS semantics, and up to 21% slower than the NFS implementation in Linux with Ext2fs, which does not implement the correct semantics.

Finally, the experiments with proactive recovery show that the overhead is low even with very frequent recoveries: it ranges from 27% with a minimum window of vulnerability of 1.5 minutes to 2% with a window of vulnerability of 10.5 minutes. Therefore, it is possible to improve resilience by decreasing the window of vulnerability with a low impact on performance.

Chapter 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 tolerate software bugs, operator mistakes, or malicious attacks. Most of this work relied on assumptions that are not realistic in the presence of these types of faults, and the work that did not rely on unrealistic assumptions performed poorly and did not provide a complete solution to build replicated systems.

Sections 9.1 and Section 9.2 discuss replication techniques that assume benign faults and replication techniques that tolerate Byzantine faults, respectively. Section 9.3 discusses other related work.

9.1 Replication With Benign Faults

Most research on replication has focused on techniques that tolerate *benign faults* (e.g., [AD76, Lam78, Gif79, OL88, Lam89, LGG$^+$91]): they assume 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: the probability of incorrect system behavior increases with the number of replicas.

Viewstamped replication [OL88] and Paxos [Lam89] use a combination of primary-backup [AD76] and quorum [Gif79] 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: BFT uses cryptographic authentication, quorum certificates, an extra pre-prepare phase, and different techniques to perform view changes, select primaries, and garbage collect information.

We are the first to provide a replicated file system that tolerates Byzantine faults but there are several replicated file systems that tolerate benign faults, e.g. Ficus [GHM+90], Coda [Sat90], Echo [HBJ+90], and Harp [LGG+91]. Our system is most similar to Harp, which also implements the NFS protocol. Like Harp, we take advantage of replication to ensure stability of modified data and meta-data before replying to clients (as required by the NFS protocol) without synchronous disk writes.

9.2 Replication With Byzantine Faults

Techniques that tolerate *Byzantine faults* [PSL80, LSP82] make no assumptions about the behavior of faulty components and, therefore, can tolerate even malicious attacks. However, most earlier work (e.g., [PSL80, LSP82, Sch90, CASD85, Rei96, MR96a, GM98, KMMS98]) assumes synchrony, which is not a good assumption in real systems because of bursty load in both the processors and the network. This assumption is particularly dangerous with malicious attackers that can launch denial-of-service attacks to flood the processors or the network with spurious requests.

Agreement and Consensus

Some agreement and consensus algorithms tolerate Byzantine faults in asynchronous systems (e.g, [BT85, CR92, MR96b, DGGS99, CKS00]). However, they do not provide a complete solution for state machine replication, and furthermore, most of them were designed to demonstrate theoretical feasibility and are too slow to be used in practice.

BFT's protocol during normal-case operation is similar to the Byzantine agreement algorithm in [BT85]. 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. Their algorithm also uses symmetric cryptography but since it does not provide view changes, garbage collection, or client authentication, it does not solve the problems that make eliminating public-key cryptography hard.

The algorithm in [CKS00] solves consensus more efficiently than previous algorithms. It is possible to use this algorithm as a building block to implement state machine replication but the performance would be poor: it would require 7 message delays to process client requests and it would perform at least three public-key signatures in the critical path. The algorithm in [CKS00] uses a signature sharing scheme to generate the equivalent of our quorum certificates. This is interesting: it could be combined with proactive signature sharing [HJJ+97] to produce certificates that could be exchanged among replicas even with recoveries.

State Machine Replication

Our work is inspired by Rampart [Rei94, Rei95, Rei96, MR96a] and SecureRing [KMMS98], which also implement state machine replication. However, these systems rely on synchrony assumptions for safety.

Both Rampart and SecureRing use group communication techniques [BSS91] 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 [Lyn96], i.e., they may misclassify a replica as faulty. Since correctness requires that fewer than $1/3$ of group members be faulty, a misclassification can compromise correctness by removing a non-faulty replica from the group. This opens an avenue of attack: an attacker gains control over a single replica but does not change its behavior in any detectable way; then it slows correct replicas or the communication between them until enough are excluded from the group. It is even possible for these system 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 loosing security.

Rampart and SecureRing can guarantee safety only if fewer than $1/3$ of the replicas are faulty during the lifetime of the system. This guarantee is too weak for long-lived systems. Our system improves this guarantee by recovering replicas proactively and frequently; it can tolerate any number of faults if fewer than $1/3$ of the replicas become faulty within a window of vulnerability, which can be made small under normal load conditions with low impact on performance.

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. First, the system may be unable to provide safety if a replica that is not faulty is removed from the group to be recovered. Second, the algorithms rely on messages signed by replicas even after they are removed from the group and there is no way to prevent attackers from impersonating removed replicas that they controlled.

Quorum Replication

Phalanx [MR97, MR98a, MR98b] and its successor Fleet [MR00] apply quorum replication techniques [Gif79] 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 to acquire locks. We can implement arbitrary operations that access any number of variables and can both read and write to those variables, whereas in Fleet it would be necessary to acquire and release locks to execute such operations. This makes Fleet more vulnerable to malicious clients because it relies on clients to group and order reads and blind writes to preserve any invariants over the service state.

Fleet provides an algorithm with optimal resilience ($n > 3f + 1$ 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 $n > 4f + 1$ 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 [APMR99] and a mechanism to adapt the threshold f on the number of faults tolerated by the system based on this estimate [AMP$^+$00]. 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 other 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 and reads require one or two round-trips. Our algorithm executes read-write operations in two round-trips and most read-only operations in one. Furthermore, 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 partitioning the state by several state machine replica groups is a better approach to achieve scalability for most applications. Furthermore, it is possible to combine our algorithm with quorum systems that tolerate benign faults to improve on Fleet's scalability but this is future work.

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 state transfer algorithm is also unusual because it is highly asynchronous. In replication algorithms for benign faults, e.g.,[LGG$^+$91], replicas typically retain a checkpoint of the state and messages in their log until the recovering replica is brought up-to-date. This could open an avenue for a denial-of-service attack in the presence of Byzantine faults. Instead, in our algorithm, replicas are free to garbage collect information and are minimally delayed by the recovery.

The SFS read-only file system [FKM00] can tolerate Byzantine faults. This file system 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 [Mer87] 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. Our technique to check the integrity of the replica's state during recovery is similar to those in [BEG$^+$94] and [MVS00] except that we obtain the tree with correct digests from the other replicas rather than from a secure co-processor.

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 [OY91]. This concept has previously been applied in synchronous systems to secret-sharing schemes [HJKY95], threshold cryptography [HJJ$^+$97], and more recently secure information storage and retrieval [GGJR99] (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.

Chapter 10

Conclusions

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. This thesis describes a new replication algorithm and implementation techniques to build highly-available systems that tolerate Byzantine faults. It shows, for the first time, how to build Byzantine-fault-tolerant systems that can be used in practice to implement real services because they do not rely on unrealistic assumptions and they perform well.

This chapter presents a summary of the main results in the thesis and directions for future work.

10.1 Summary

This thesis describes BFT, a state-machine replication algorithm that tolerates Byzantine faults provided fewer than $1/3$ of the replicas are faulty.

BFT does not rely on unrealistic assumptions. For example, it is bad to assume synchrony because a denial-of-service attack can cause the service to return incorrect replies. BFT is the first state-machine replication algorithm that works correctly in asynchronous systems with Byzantine faults: it provides linearizability, which is a strong safety property, without relying on any synchrony assumption. Additionally, it guarantees liveness provided message delays are bounded eventually. A service may be unable to return replies when a denial of service attack is active but it never returns incorrect replies and clients are guaranteed to receive replies when the attack ends.

It is also bad to assume that client faults are benign because clients are usually easier to compromise than replicas. BFT provides safety and liveness regardless of the number of Byzantine-faulty clients. Additionally, it can be used to replicate services with complex operations, which is important to limit the damage Byzantine-faulty clients can cause. Service operations can be designed to preserve invariants on the service state and to perform access control; BFT ensures faulty clients are unable to break these invariants or bypass the access control checks. Algorithms that restrict service operations to simple reads and blind writes are more vulnerable to Byzantine-faulty clients because they rely on the clients to order and group these simple operations correctly in order to enforce invariants.

It is not realistic to assume that fewer than $1/3$ of the replicas fail over the lifetime of the system. This thesis 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. This mechanism recovers replicas periodically even if there is no reason to suspect that they are faulty. Replicas can be recovered frequently to shrink the window of vulnerability to a few minutes with a low impact on performance. Additionally, the proactive recovery 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 has been implemented as a generic program library with a simple interface. The BFT library provides a complete solution to the problem of building real services that tolerate Byzantine faults. For example, it includes efficient techniques to garbage collect information, to transfer state to bring replicas up-to-date, to retransmit messages, and to handle services with non-deterministic behavior. The thesis describes a real service that was implemented using the BFT library: the first Byzantine-fault-tolerant NFS file system, BFS.

The BFT library and BFS perform well. For example, BFS 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. 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 over many requests. The algorithm also uses optimizations to reduce protocol overhead as the operation argument and return sizes increase.

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. 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 [Gra00] since they are the hardest to detect. In fact, we encountered such a software bug while running our system, and our algorithm was able to continue running correctly in spite of it. The BFT library can also mask software errors due to aging (e.g., resource leaks). It improves on the usual technique of rebooting the system because it refreshes state automatically and staggers recovery so that individual replicas are highly unlikely to fail simultaneously. Additionally, systems replicated with the BFT library can tolerate attacks that take longer than the window of vulnerability to succeed.

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.

An agent from Europol reported in a recent news article [Sul00] that a bank lost millions of dollars through a scheme implemented by one of its own system administrators who added a few lines of code to the bank's software. The BFT library could have prevented this problem.

10.2 Future Work

We want to explore the use of software diversity to improve resilience to software bugs and attacks that exploit software bugs because these faults are the most common. N-version programming [CA78] is expensive but since there are several independent implementations available of operating systems and important services (e.g., file systems, data bases, and WEB servers), replicas can run different operating systems and different implementations of the code for these services. For this to work, it is necessary to implement a small software layer to ensure that the different replicas have the same *observable behavior*. This is simplified by the existence of standardized protocols to access important services (e.g., NFS [S^+85] and ODBC [Gei95]) but there are some interesting issues on how to implement this layer efficiently.

Additionally, for checkpoint management and state transfer to work with software diversity, it is necessary to define a common *observable service state* and to implement efficient translation functions between the state in each implementation and this observable state. Since the observable state abstracts away implementation details, this technique will also improve resilience to resource leaks in the service code; our state transfer technique can be used to restart a replica from a correct checkpoint of the observable state that is obtained from the others.

It is possible to improve security further by exploiting software diversity across recoveries. One possibility is to restrict the service interface at a replica after its state is found to be corrupt. Another potential approach is to use obfuscation and randomization techniques [CT00, F^+97] to produce a new version of the software each time a replica is recovered. These techniques are not very resilient to attacks but they can be very effective when combined with proactive recovery because the attacker has a bounded time to break them.

The algorithm described in this thesis uses a fixed group of replicas. We would like to extend it to allow dynamic configuration changes. This is hard with Byzantine faults: an attacker that controls a quorum of the replicas in some old configuration may fool clients into believing that the current configuration is an arbitrary set of replicas under its control. We believe it is possible to use proactive signature sharing [HJJ^+97] to solve this problem. The idea is that the members of the group would be able to generate a shared signature that could be verified with a constant, well-known public key. Such a signature could be used to convince the clients of the current group membership. To prevent an attacker from learning how to generate a valid signature, the shares used to generate it would be refreshed on every configuration change. For this to work, it would be necessary to develop a refreshment protocol for the shares that worked both correctly and efficiently in asynchronous systems.

Another problem of special interest is reducing the amount of resources required to implement a replicated service. The number of replicas can be reduced by using f replicas as witnesses [Par86, LGG+91] that are involved in the protocol only when some full replica fails. It is also possible to reduce the number of copies of the state to $f + 1$ but the details remain to be worked out.

We have shown how to implement a Byzantine-fault-tolerant file system. It would be interesting to use the BFT library to implement other services, for example, a relational database or an `httpd`. The library has already been used to replicate the Thor [LAC+96, CALM97] object-oriented database [Rod00] and a Domain Name Service (DNS) [TPRZ84] with dynamic updates [Ahm00, Yan99]. DNS is interesting because it uses hierarchical state partitioning and caching to achieve scalability. To implement a Byzantine-fault-tolerant DNS, we had to develop an efficient protocol for replicated clients that allows the replicas in a group to request operations from another group of replicas.

This thesis has focused on the performance of the BFT library in the normal case. It is important to perform an experimental evaluation of the reliability and performance of the library with faults by using fault-injection techniques. The challenge is that attacks are hard to model. For example, attacks can involve cooperation between faulty clients and replicas, and can combine denial-of-service with penetration. Ultimately, we would like to make a replicated service available on the Internet and launch a challenge to break it.

Source Code Availability

We made the source code for the BFT library, BFS, and the benchmarks used in their performance evaluation available to allow others to reproduce our results and improve on this work. It can be obtained from:

`http://www.pmg.lcs.mit.edu/~castro/byz.html`

Appendix A

Formal Safety Proof for BFT-PK

This appendix presents a formal safety proof for the BFT-PK algorithm. The proof is based on invariant assertions and simulation relations. It shows that the algorithm A_{gc} formalized in Section 2.4 implements the automaton S, which specifies safe behavior and was defined in Section 2.4.3. We use the following strategy to show this. We start by proving that a simplified version of the algorithm, A, which does not have garbage collection, implements S. Then, we prove that A_{gc} implements A.

A.1 Algorithm Without Garbage Collection

This section specifies the simplified algorithm A, which does not have garbage collection. The proxy and multicast channel automata in A are identical to the ones defined for A_{gc} in Section 2.4. The difference is in the specification of the replica automata. Each replica automaton R_i in A is defined as follows.

Signature:

Input:	$\text{RECEIVE}(\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})_i$
	$\text{RECEIVE}(\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j})_i$
	$\text{RECEIVE}(\langle \text{PREPARE}, v, n, d, j\rangle_{\sigma_j})_i$
	$\text{RECEIVE}(\langle \text{COMMIT}, v, n, d, j\rangle_{\sigma_j})_i$
	$\text{RECEIVE}(\langle \text{VIEW-CHANGE}, v, P, j\rangle_{\sigma_j})_i$
	$\text{RECEIVE}(\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$
	REPLICA-FAILURE_i
Internal:	$\text{SEND-PRE-PREPARE}(m, v, n)_i$
	$\text{SEND-COMMIT}(m, v, n)_i$
	$\text{EXECUTE}(m, v, n)_i$
	$\text{VIEW-CHANGE}(v)_i$
	$\text{SEND-NEW-VIEW}(v, V)_i$
Output:	$\text{SEND}(m, X)_c$

Here, $t, v, n \in \mathbf{N}$, $c \in \mathcal{C}$, $i, j \in \mathcal{R}$, $m \in \mathcal{M}$, $V, O, N \subseteq \mathcal{M}$, $X \subseteq \mathcal{X}$, and $d \in \mathcal{D} = \{d \mid \exists m \in \mathcal{M} : (d = D(m))\}$

State:

$val_i \in \mathcal{S}$, initially s_o
$view_i \in \mathbf{N}$, initially 0
$in_i \subseteq \mathcal{M}$, initially $\{\}$
$out_i \subseteq \mathcal{M}$, initially $\{\}$
$\mathit{last\text{-}rep}_i : \mathcal{C} \rightarrow \mathcal{O}'$, initially $\forall\, c \in \mathcal{C} : \mathit{last\text{-}rep}_i(c) = \mathit{null\text{-}rep}$
$\mathit{last\text{-}rep\text{-}t}_i : \mathcal{C} \rightarrow \mathbf{N}$, initially $\forall\, c \in \mathcal{C} : \mathit{last\text{-}rep\text{-}t}_i(c) = 0$
$seqno_i \in \mathbf{N}$, initially 0
$\mathit{last\text{-}exec}_i \in \mathbf{N}$, initially 0
$faulty_i \in Bool$, initially *false*

Auxiliary functions:

$tag(m,u) \equiv m = \langle u, ...\rangle$
$primary(v) \equiv v \mod |\mathcal{R}|$
$primary(i) \equiv view_i \mod |\mathcal{R}|$
$\mathit{in\text{-}v}(v,i) \equiv view_i = v$
$prepared(m,v,n,M) \equiv \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_{primary(v)}} \in M \wedge$
$\quad \exists\, R : (|R| \geq 2f \wedge primary(v) \notin R \wedge \forall\, k \in R : (\langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} \in M))$
$prepared(m,v,n,i) \equiv prepared(m,v,n,in_i)$
$\mathit{last\text{-}prepared}(m,v,n,M) \equiv prepared(m,v,n,M) \wedge$
$\quad \nexists\, m', v' : ((prepared(m',v',n,M) \wedge v' > v) \vee (prepared(m',v,n,M) \wedge m \neq m'))$
$\mathit{last\text{-}prepared}(m,v,n,i) \equiv \mathit{last\text{-}prepared}(m,v,n,in_i)$
$committed(m,v,n,i) \equiv (\exists\, v' : (\langle \text{PRE-PREPARE}, v', n, m\rangle_{\sigma_{primary(v')}} \in in_i) \vee m \in in_i) \wedge$
$\quad \exists\, R : (|R| \geq 2f+1 \wedge \forall\, k \in R : (\langle \text{COMMIT}, v, n, D(m), k\rangle_{\sigma_k} \in in_i))$
$\mathit{correct\text{-}view\text{-}change}(m,v,j) \equiv \exists\, P : (m = \langle \text{VIEW-CHANGE}, v, P, j\rangle_{\sigma_j} \wedge$
$\forall\, \langle \text{PRE-PREPARE}, v', n, m'\rangle_{\sigma_{primary(v')}} \in P : (\mathit{last\text{-}prepared}(m',v',n,P) \wedge v' < v)$
$\mathit{merge\text{-}P}(V) \equiv \{\, m \mid \exists\, \langle \text{VIEW-CHANGE}, v, P, k\rangle_{\sigma_k} \in V : m \in P \,\}$
$\mathit{max\text{-}n}(M) \equiv max(\{\, n \mid \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i} \in M\})$
$\mathit{correct\text{-}new\text{-}view}(m,v) \equiv$
$\quad \exists\, V,O,N,R : (m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_{primary(v)}} \wedge |V| = |R| = 2f+1 \wedge$
$\quad \forall\, k \in R : (\exists\, m' \in V : (\mathit{correct\text{-}view\text{-}change}(m',v,k))) \wedge$
$\quad O = \{\, \langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_{primary(v)}} \mid \exists\, v' : \mathit{last\text{-}prepared}(m',v',n,\mathit{merge\text{-}P}(V))\} \wedge$
$\quad N = \{\, \langle \text{PRE-PREPARE}, v, n, null\rangle_{\sigma_{primary(v)}} \mid n < \mathit{max\text{-}n}(O) \wedge$
$\qquad \nexists\, v', m', n : \mathit{last\text{-}prepared}(m',v',n,\mathit{merge\text{-}P}(V)))$
$\mathit{has\text{-}new\text{-}view}(v,i) \equiv v = 0 \vee \exists\, m : (m \in in_i \wedge \mathit{correct\text{-}new\text{-}view}(m,v))$

Output Transitions:

$\text{SEND}(m, \mathcal{R} - \{i\})_i$
Pre: $m \in out_i \wedge \neg tag(m, \text{REQUEST}) \wedge \neg tag(m, \text{REPLY})$
Eff: $out_i := out_i - \{m\}$

$\text{SEND}(m, \{primary(i)\})_i$
Pre: $m \in out_i \wedge tag(m, \text{REQUEST})$
Eff: $out_i := out_i - \{m\}$

$\text{SEND}(\langle \text{REPLY}, v, t, c, i, r\rangle_{\sigma_i}, \{c\})_i$
Pre: $\langle \text{REPLY}, v, t, c, i, r\rangle_{\sigma_i} \in out_i$
Eff: $out_i := out_i - \{\langle \text{REPLY}, v, t, c, i, r\rangle_{\sigma_i}\}$

Input Transitions:

RECEIVE($\langle$REQUEST$, o, t, c\rangle_{\sigma_c})_i$
Eff: let $m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c}$
if $t = \textit{last-rep-t}_i(c)$ then
$\textit{out}_i := \textit{out}_i \cup \{\langle$REPLY$, \textit{view}_i, t, c, i, \textit{last-rep}_i(c)\rangle_{\sigma_i}\}$
else
$\textit{in}_i := \textit{in}_i \cup \{m\}$
if $\textit{primary}(i) \neq i$ then
$\textit{out}_i := \textit{out}_i \cup \{m\}$

RECEIVE($\langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: if $j = \textit{primary}(i) \land \textit{in-v}(v, i) \land \textit{has-new-view}(v, i) \land$
$\nexists d : (d \neq D(m) \land \langle$PREPARE$, v, n, d, i\rangle_{\sigma_i} \in \textit{in}_i)$ then
let $p = \langle$PREPARE$, v, n, D(m), i\rangle_{\sigma_i}$
$\textit{in}_i := \textit{in}_i \cup \{\langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_j}, p\}$
$\textit{out}_i := \textit{out}_i \cup \{p\}$
else if $\exists o, t, c : (m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c})$ then
$\textit{in}_i := \textit{in}_i \cup \{m\}$

RECEIVE($\langle$PREPARE$, v, n, d, j\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: if $j \neq \textit{primary}(i) \land \textit{in-v}(v, i)$ then
$\textit{in}_i := \textit{in}_i \cup \{\langle$PREPARE$, v, n, d, j\rangle_{\sigma_j}\}$

RECEIVE($\langle$COMMIT$, v, n, d, j\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: if $\textit{view}_i \geq v$ then
$\textit{in}_i := \textit{in}_i \cup \{\langle$COMMIT$, v, n, d, j\rangle_{\sigma_j}\}$

RECEIVE($\langle$VIEW-CHANGE$, v, \mathcal{P}, j\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: let $m = \langle$VIEW-CHANGE$, v, \mathcal{P}, j\rangle_{\sigma_j}$
if $v \geq \textit{view}_i \land \textit{correct-view-change}(m, v, j)$ then
$\textit{in}_i := \textit{in}_i \cup \{m\}$

RECEIVE($\langle$NEW-VIEW$, v, X, O, N\rangle_{\sigma_j})_i$ $(j \neq i)$
Eff: let $m = \langle$NEW-VIEW$, v, X, O, N\rangle_{\sigma_j}$,
$P = \{\langle$PREPARE$, v, n', D(m'), i\rangle_{\sigma_i} | \langle$PRE-PREPARE$, v, n', m'\rangle_{\sigma_j} \in (O \cup N)\}$
if $v > 0 \land v \geq \textit{view}_i \land \textit{correct-new-view}(m, v) \land \neg\textit{has-new-view}(v, i)$ then
$\textit{view}_i := v$
$\textit{in}_i := \textit{in}_i \cup O \cup N \cup \{m\} \cup P$
$\textit{out}_i := P$

REPLICA-FAILURE$_i$
Eff: $\textit{faulty}_i := \textit{true}$

Internal Transitions:

SEND-PRE-PREPARE$(m, v, n)_i$
Pre: $\textit{primary}(i) = i \land \textit{seqno}_i = n - 1 \land \textit{in-v}(v, i) \land \textit{has-new-view}(v, i) \land$
$\exists o, t, c : (m = \langle$REQUEST$, o, t, c\rangle_{\sigma_c} \land m \in \textit{in}_i) \land \nexists\langle$PRE-PREPARE$, v, n', m\rangle_{\sigma_i} \in \textit{in}_i$
Eff: $\textit{seqno}_i := \textit{seqno}_i + 1$
let $p = \langle$PRE-PREPARE$, v, n, m\rangle_{\sigma_i}$
$\textit{out}_i := \textit{out}_i \cup \{p\}$
$\textit{in}_i := \textit{in}_i \cup \{p\}$

SEND-COMMIT$(m, v, n)_i$
Pre: $\textit{prepared}(m, v, n, i) \land \langle$COMMIT$, v, n, D(m), i\rangle_{\sigma_i} \notin \textit{in}_i$
Eff: let $c = \langle$COMMIT$, v, n, D(m), i\rangle_{\sigma_i}$
$\textit{out}_i := \textit{out}_i \cup \{c\}$
$\textit{in}_i := \textit{in}_i \cup \{c\}$

$\text{EXECUTE}(m, v, n)_i$
Pre: $n = \textit{last-exec}_i + 1 \wedge \textit{committed}(m, v, n, i)$
Eff: $\textit{last-exec}_i := n$
if $(m \neq \textit{null})$ then
let $\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c} = m$
if $t \geq \textit{last-rep-t}_i(c)$ then
if $t > \textit{last-rep-t}_i(c)$ then
$\textit{last-rep-t}_i(c) := t$
$(\textit{last-rep}_i(c), \textit{val}_i) := g(c, o, \textit{val}_i)$
$\textit{out}_i := \textit{out}_i \cup \{\langle \text{REPLY}, \textit{view}_i, t, c, i, \textit{last-rep}_i(c)\rangle_{\sigma_i}\}$
$\textit{in}_i := \textit{in}_i - \{m\}$

$\text{SEND-VIEW-CHANGE}(v)_i$
Pre: $v = \textit{view}_i + 1$
Eff: $\textit{view}_i := v$
let $P' = \{\langle m, v, n\rangle | \textit{last-prepared}(m, v, n, i)\}$,
$P = \bigcup_{\langle m,v,n\rangle \in P'} (\{p = \langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} | p \in \textit{in}_i\} \cup \{\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_{\textit{primary}(v)}}\})$,
$m = \langle \text{VIEW-CHANGE}, v, P, i\rangle_{\sigma_i}$
$\textit{out}_i := \textit{out}_i \cup \{m\}$
$\textit{in}_i := \textit{in}_i \cup \{m\}$

$\text{SEND-NEW-VIEW}(v, V)_i$
Pre: $\textit{primary}(v) = i \wedge v \geq \textit{view}_i \wedge v > 0 \wedge V \subseteq \textit{in}_i \wedge |V| = 2f + 1 \wedge \neg \textit{has-new-view}(v, i) \wedge$
$\exists R : (|R| = 2f + 1 \wedge \forall k \in R : (\exists P : (\langle \text{VIEW-CHANGE}, v, P, k\rangle_{\sigma_k} \in V)))$
Eff: $\textit{view}_i := v$
let $O = \{\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i} | \exists v' : \textit{last-prepared}(m, v', n, \textit{merge-P}(V))\}$,
$N = \{\langle \text{PRE-PREPARE}, v, n, \textit{null}\rangle_{\sigma_i} | n < \textit{max-n}(O) \wedge \nexists v', m, n : \textit{last-prepared}(m, v', n, \textit{merge-P}(V))\}$,
$m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_i}$
$\textit{seqno}_i := \textit{max-n}(O)$
$\textit{in}_i := \textit{in}_i \cup O \cup N \cup \{m\}$
$\textit{out}_i := \{m\}$

Safety Proof

Next, we prove that A implements S. We start by proving some invariants. The first invariant says that messages, which are signed by a non-faulty replica, are in the replica's log. This invariant is important because its proof is the only place where it is necessary to reason about the security of signatures and it enables most of the other invariants to reason only about the local state of a replica.

The key results are Invariant A.1.4, which says that correct replicas never prepare distinct requests with the same view and sequence number, and Invariant A.1.11, which says that correct replicas never commit distinct requests with the same sequence number. We use these invariants and a simulation relation to prove that A implements S.

Invariant A.1.1 *The following is true of any reachable state in an execution of A,*

$$\begin{aligned}
&\forall i, j \in \mathcal{R}, m \in \mathcal{M} : ((\neg \textit{faulty}_i \wedge \neg \textit{faulty}_j \wedge \neg \textit{tag}(m, \text{REPLY})) \Rightarrow \\
&\quad (((\langle m\rangle_{\sigma_i} \in \textit{in}_j \vee \exists\, m' = \langle \text{VIEW-CHANGE}, v, P, k\rangle_{\sigma_k} : (m' \in \textit{in}_j \wedge \langle m\rangle_{\sigma_i} \in P) \vee \\
&\quad \exists\, m' = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_k} : (m' \in \textit{in}_j \wedge (\langle m\rangle_{\sigma_i} \in V \vee \langle m\rangle_{\sigma_i} \in \textit{merge-P}(V)))) \\
&\quad\quad \Rightarrow \langle m\rangle_{\sigma_i} \in \textit{in}_i))
\end{aligned}$$

The same is also true if one replaces in_j *by* $\{m \mid \exists X : (m, X) \in \text{wire}\}$ *or by* out_j

Proof: For any reachable state x of A and message value m that is not a reply message, if replica i is not faulty in state x, $\langle m \rangle_{\sigma_i} \in out_i \Rightarrow \langle m \rangle_{\sigma_i} \in in_i$. Additionally, if $\langle m \rangle_{\sigma_i} \in in_i$ is true for some state in an execution, it remains true in all subsequent states in that execution or until i becomes faulty. By inspection of the code for automaton R_i, these two conditions are true because every action of R_i that inserts a message $\langle m \rangle_{\sigma_i}$ in out_i also inserts it in in_i and no action ever removes a message signed by i from in_i.

Our assumption on the strength of authentication guarantees that no automaton can impersonate a non-faulty replica R_i by sending $\langle m \rangle_{\sigma_i}$ (for all values of m) on the multicast channel. Therefore, for a signed message $\langle m \rangle_{\sigma_i}$ to be in some state component of a non-faulty automaton other than R_i, it is necessary for $\text{SEND}(\langle m \rangle_{\sigma_i}, X)_i$ to have executed for some value of X at some earlier point in that execution. The precondition for the execution of such a send action requires $\langle m \rangle_{\sigma_i} \in out_i$. The latter and the two former conditions prove the invariant. ∎

The next batch of invariants states self-consistency conditions for the state of individual replicas. For example, it states that replicas never log conflicting pre-prepare or prepare messages for the same view and sequence number.

Invariant A.1.2 *The following is true of any reachable state in an execution of A, for any replica i such that faulty$_i$ is* false*:*

1. $\forall \langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i} \in in_i : (\nexists d' \neq d : (\langle \text{PREPARE}, v, n, d', i \rangle_{\sigma_i} \in in_i))$
2. $\forall v, n, m : ((i = primary(v) \wedge \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_i} \in in_i) \Rightarrow$
   $\nexists m' : (m' \neq m \wedge \langle \text{PRE-PREPARE}, v, n, m' \rangle_{\sigma_i} \in in_i))$
3. $\forall \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_i} \in in_i : (i = primary(v) \Rightarrow n \leq seqno_i)$
4. $\forall \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_{primary(v)}} \in in_i :$
   $(v > 0 \Rightarrow \exists m' = \langle \text{NEW-VIEW}, v, X, O, N \rangle_{\sigma_{primary(v)}} : (m' \in in_i \wedge correct\text{-}new\text{-}view(m', v)))$
5. $\forall m' = \langle \text{NEW-VIEW}, v, X, O, N \rangle_{\sigma_{primary(v)}} \in in_i : correct\text{-}new\text{-}view(m', v)$
6. $\forall m' = \langle \text{VIEW-CHANGE}, v, \mathcal{P}, j \rangle_{\sigma_j} \in in_i : correct\text{-}view\text{-}change(m', v, j)$
7. $\forall \langle \text{PREPARE}, v, n, D(m), i \rangle_{\sigma_i} \in in_i : (\langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_{primary(v)}} \in in_i)$
8. $\forall \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_{primary(v)}} \in in_i : (i \neq primary(v) \Rightarrow \langle \text{PREPARE}, v, n, D(m), i \rangle_{\sigma_i} \in in_i)$
9. $\forall \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_{primary(v)}} \in in_i : v \leq view_i$

Proof: The proof is by induction on the length of the execution. The initializations ensure that $in_i = \{\}$ and, therefore, all conditions are vacuously true in the base case. For the inductive step, assume that the invariant holds for every state of any execution α of length at most l. We will show that the invariant also holds for any one step extension α_1 of α.

Condition (1) can be violated in α_1 only if an action that may insert a prepare message signed by i in in_i executes. These are actions of the form:

1. $\text{RECEIVE}(\langle \text{PRE-PREPARE}, v, n, m' \rangle_{\sigma_j})_i$
2. $\text{RECEIVE}(\langle \text{PREPARE}, v, n, d, j \rangle_{\sigma_j})_i$
3. $\text{RECEIVE}(\langle \text{NEW-VIEW}, v, V, O, N \rangle_{\sigma_j})_i$

The first type of action cannot violate condition (1) because the condition in the *if* statement ensures that $\langle\text{PREPARE}, v, n, D(m'), i\rangle_{\sigma_i}$ is not inserted in in_i when there exists a $\langle\text{PREPARE}, v, n, d, i\rangle_{\sigma_i} \in in_i$ such that $D(m') \neq d$. Similarly, the second type of action cannot violate condition (1) because it only inserts the argument prepare message in in_i if it is signed by a replica other than R_i.

For the case $v = 0$, actions of type 3 never have effects on the state of R_i. For the case $v > 0$, we can apply the inductive hypothesis of conditions (7) and (4) to conclude that if there existed a $\langle\text{PREPARE}, v, n, D(m), i\rangle_{\sigma_i} \in in_i$ in the last state in α, there would also exist a new-view message for view v in in_i in that state. Therefore, the precondition of actions of type 3 would prevent them from executing in such a state. Since actions of type 3 may insert multiple prepare messages signed by R_i into in_i, there is still a chance they can violate condition (1). However, this cannot happen because these actions are enabled only if the argument new-view message is correct and the definition of *correct-new-view* ensures that there is at most one pre-prepare message with a given sequence number in $O \cup N$.

Condition (2) can be violated in α_1 only by the execution of an action of one of the following types:

1. $\text{RECEIVE}(\langle\text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_j})_i$,
2. $\text{RECEIVE}(\langle\text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$,
3. $\text{SEND-PRE-PREPARE}(m, v, n)_i$, or
4. $\text{SEND-NEW-VIEW}(v, V)_i$

Actions of the first two types cannot violate condition (2) because they only insert pre-prepare messages in in_i that are not signed by R_i. Actions of the third type cannot violate condition (2) because the inductive hypothesis for condition (3) and the precondition for the send-pre-prepare action ensure that the pre-prepare message inserted in in_i has a sequence number that is one higher than the sequence number of any pre-prepare message for the same view signed by R_i in in_i. Finally, actions of the fourth type cannot violate condition (2). For $v = 0$, they are not enabled. For $v > 0$, the inductive hypothesis of condition (4) and the precondition for the send-new-view action ensure that no pre-prepare for view v can be in in_i when the action executes, and the definition of O and N ensures that there is at most one pre-prepare message with a given sequence number in $O \cup N$.

Condition (3) can potentially be violated by actions that insert pre-prepares in in_i or modify $seqno_i$. These are exactly the actions of the types listed for condition (2). As before, actions of the first two types cannot violate condition (3) because they only insert pre-prepare messages in in_i that are not signed by R_i and they do not modify $seqno_i$. The send-pre-prepare action preserves condition (3) because it increments $seqno_i$ such that it becomes equal to the sequence number of the pre-prepare message it inserts in in_i. The send-new-view actions also preserve condition (3): (as shown before) actions of this type only execute if there is no pre-prepare for view v in in_i and, when they execute, they set $seqno_i := \textit{max-n}(O)$, which is equal to the sequence number of the pre-prepare for view v with the highest sequence number in in_i.

To violate condition (4), an action must either insert a pre-prepare message in in_i or remove a new-view message from in_i. No action ever removes new-view messages from in_i. The actions that may insert pre-prepare messages in in_i are exactly the actions of the types listed for condition (2). The first type of action in this list cannot violate condition (4) because the *if* statement in its body ensures that the argument pre-prepare message is inserted in in_i only when $\textit{has-new-view}(v, i)$ is true. The second type of action only inserts pre-prepare messages for view v in in_i if the argument new-view message is correct and in this case it also inserts the argument new-view message in in_i. Therefore, the second type of action also preserves condition (4). The precondition of send-pre-prepare actions ensures that send-pre-prepare actions preserve condition (4). Finally, the send-new-view actions also preserve condition (4) because their effects and the inductive hypothesis for condition (6) ensure that a correct new-view message for view v is inserted in in_i whenever a pre-prepare for view v is inserted in in_i.

Conditions (5) and (6) are never violated. First, received new-view and view-change messages are always checked for correctness before being inserted in in_i. Second, the effects of send-view-change actions together with the inductive hypothesis of condition (9) and the precondition of send-view-change actions ensure that only correct view-change messages are inserted in in_i. Third, the inductive hypothesis of condition (6) and the effects of send-new-view actions ensure that only correct new-view messages are inserted in in_i.

Condition (7) is never violated because no action ever removes a pre-prepare from in_i and the actions that insert a $\langle \text{PREPARE}, v, n, D(m), i\rangle_{\sigma_i}$ in in_i (namely $\text{RECEIVE}(\langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_j})_i$ and $\text{RECEIVE}(\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$ actions) also insert a $\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_{primary(v)}}$ in in_i.

Condition (8) can only be violated by actions that insert pre-prepare messages in in_i because prepare messages are never removed from in_i. These are exactly the actions listed for condition (2). The first two types of actions preserve condition (8) because whenever they insert a pre-prepare message in in_i they always insert a matching prepare message. The last two types of actions can not violate condition (8) because they never insert pre-prepare messages for views v such that $primary(v) \neq i$ in in_i.

The only actions that can violate condition (9) are actions that insert pre-prepare messages in in_i or make $view_i$ smaller. Since no actions ever make $view_i$ smaller, the actions that may violate condition (9) are exactly those listed for condition (2). The *if* statement in the first type of action ensures that it only inserts pre-prepare messages in in_i when their view number is equal to $view_i$. The *if* statement in the second type of action ensures that it only inserts pre-prepare messages in in_i when their view number is greater than or equal to $view_i$. Therefore, both types of actions preserve the invariant. The precondition for the third type of action and the effects of the fourth type of action ensure that only pre-prepare messages with view number equal to $view_i$ are inserted in in_i. Thus, these two types of actions also preserve the invariant. □

Definition A.1.3 n-faulty $\equiv |\{i \in \mathcal{R} | \text{faulty}_i = \text{true}\}|$

The next two invariants are important. They state that replicas agree on an order for requests within a single view, i.e., it is impossible to produce prepared certificates with the same view and sequence number and with distinct requests. The intuition behind the proof is that correct replicas do not accept conflicting pre-prepare messages with the same view and sequence number, and that the quorums corresponding to any two certificates intersect in at least one correct replica.

Invariant A.1.4 *The following is true of any reachable state in an execution of A,*
$\forall\, i,j \in \mathcal{R},\ n,v \in \mathbf{N},\ m,m' \in \mathcal{M} : ((\neg \textit{faulty}_i \wedge \neg \textit{faulty}_j \wedge \textit{n-faulty} \leq f) \Rightarrow$
$(\textit{prepared}(m,v,n,i) \wedge \textit{prepared}(m',v,n,j) \Rightarrow D(m) = D(m')))$

Proof: By contradiction, assume the invariant does not hold. Then $\textit{prepared}(m,v,n,i) = \textit{true}$ and $\textit{prepared}(m',v,n,j) = \textit{true}$ for some values of m, m', v, n, i, j such that $D(m') \neq D(m)$. Since there are $3f + 1$ replicas, this condition and the definition of the *prepared* predicate imply:

(a) $\exists R : (|R| > f \wedge \forall k \in R :$

$(((\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k} \in \textit{in}_i \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} \in \textit{in}_i) \wedge$

$(((\langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_k} \in \textit{in}_j \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m'), k\rangle_{\sigma_k} \in \textit{in}_j)))$

Since there are at most f faulty replicas and R has size at least $f + 1$, condition (a) implies:

(b) $\exists k \in R : (\textit{faulty}_k = \textit{false} \wedge$

$((((\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k} \in \textit{in}_i \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} \in \textit{in}_i) \wedge$

$(((\langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_k} \in \textit{in}_j \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m'), k\rangle_{\sigma_k} \in \textit{in}_j)))$

Invariant A.1.1 and (b) imply:

(c) $\exists k \in R : (\textit{faulty}_k = \textit{false} \wedge$

$((((\langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k} \in \textit{in}_k \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m), k\rangle_{\sigma_k} \in \textit{in}_k) \wedge$

$(((\langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_k} \in \textit{in}_k \wedge k = \textit{primary}(v)) \vee \langle \text{PREPARE}, v, n, D(m'), k\rangle_{\sigma_k} \in \textit{in}_k)))$

Condition (c) contradicts Invariant A.1.2 (conditions 1, 7 and 2.) □

Invariant A.1.5 *The following is true of any reachable state in an execution of A,*
$\forall\, i \in \mathcal{R} : ((\neg \textit{faulty}_i \wedge \textit{n-faulty} \leq f) \Rightarrow$
$(\forall\, \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_k} \in \textit{in}_i,\ n, v' \in \mathbf{N} :$
$(\textit{prepared}(m, v', n, \textit{merge-P}(V)) \wedge \textit{prepared}(m', v', n, \textit{merge-P}(V)) \Rightarrow D(m) = D(m'))))$

Proof: Since Invariant A.1.2 (condition 5) ensures any new-view message in $\textit{in}_i$ for a non-faulty i satisfies *correct-new-view*, the proof for Invariant A.1.4 can also be used here with minor modifications. □

Invariants A.1.6 to A.1.10 show that ordering information in prepared certificates stored by a quorum is propagated to subsequent views. The intuition is that new-view messages are built by collecting prepared certificates from a quorum and any two quorums intersect in at least one correct replica. These invariants allow us to prove Invariant A.1.11, which shows that replicas agree on the sequence numbers of committed requests.

Invariant A.1.6 *The following is true of any reachable state in an execution of A,*

$$\forall i \in \mathcal{R} : (\neg \mathit{faulty}_i \Rightarrow \forall \langle \text{COMMIT}, v, n, d, i\rangle_{\sigma_i} \in \mathit{in}_i : (\exists m : (D(m) = d \wedge \mathit{prepared}(m, v, n, i) = \mathit{true}))$$

Proof: The proof is by induction on the length of the execution. The initializations ensure that $\mathit{in}_i = \{\}$ and, therefore, the condition is vacuously true in the base case. For the inductive step, the only actions that can violate the condition are those that insert commit messages in in_i, i.e., actions of the form $\text{RECEIVE}(\langle \text{COMMIT}, v, n, d, j\rangle_{\sigma_j})_i$ or $\text{SEND-COMMIT}(m, v, n)_i$. Actions of the first type never violate the lemma because they only insert commit messages signed by replicas other than R_i in in_i. The precondition for send-commit actions ensures that they only insert $\langle \text{COMMIT}, v, n, D(m), i\rangle_{\sigma_i}$ in in_i if $\mathit{prepared}(m, v, n, i)$ is true. □

Invariant A.1.7 *The following is true of any reachable state in an execution of A,*

$$\begin{aligned}&\forall\, i \in \mathcal{R},\ n, v \in \mathbf{N},\ m \in \mathcal{M} : ((\neg \mathit{faulty}_i \wedge \mathit{committed}(m, v, n, i)) \Rightarrow \\ &\quad (\exists\, R : (|R| > 2f - \mathit{n\text{-}faulty} \wedge \forall\, k \in R : (\mathit{faulty}_k = \mathit{false} \wedge \mathit{prepared}(m, v, n, k)))))\end{aligned}$$

Proof: From the definition of the *committed* predicate $\mathit{committed}(m, v, n, i) = \mathit{true}$ implies

(a) $\exists R : (|R| \geq 2f + 1 \wedge \forall k \in R : (\langle \text{COMMIT}, v, n, D(m), k\rangle_{\sigma_k} \in \mathit{in}_i))$.

Invariant A.1.1 implies

(b) $\exists R : (|R| > 2f - \mathit{n\text{-}faulty} \wedge \forall k \in R : (\mathit{faulty}_k = false \wedge \langle \text{COMMIT}, v, n, D(m), k\rangle_{\sigma_k} \in \mathit{in}_k))$.

Invariant A.1.6 and (b) prove the invariant. □

Invariant A.1.8 *The following are true of any reachable state in an execution of A, for any replica i such that faulty$_i$ is* false*:*

1. $\forall m, v, n, P : (\langle \text{VIEW-CHANGE}, v, P, i\rangle_{\sigma_i} \in \mathit{in}_i \Rightarrow$
   $\forall v' < v : (\mathit{last\text{-}prepared\text{-}b}(m, v', n, i, v) \Leftrightarrow \mathit{last\text{-}prepared}(m, v', n, P)))$
2. $\forall m = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_{\mathit{primary}(v)}} \in \mathit{in}_i : ((O \cup N) \subseteq \mathit{in}_i)$

Where *last-prepared-b* is defined as follows:

$$\begin{aligned}&\mathit{last\text{-}prepared\text{-}b}(m, v, n, i, b) \equiv v < b \wedge \mathit{prepared}(m, v, n, \mathit{in}_i) \wedge \\ &\quad \nexists m', v' : ((\mathit{prepared}(m', v', n, \mathit{in}_i) \wedge v < v' < b) \vee (\mathit{prepared}(m', v, n, \mathit{in}_i) \wedge m \neq m')).\end{aligned}$$

Proof: The proof is by induction on the length of the execution. The initializations ensure that $\mathit{in}_i = \{\}$ and, therefore, the condition is vacuously true in the base case.

For the inductive step, the only actions that can violate condition (1) are those that insert view-change messages in in_i and those that insert pre-prepare or prepare messages in in_i (no pre-prepare or prepare message is ever removed from in_i.)

These actions have one of the following schemas:

1. $\text{RECEIVE}(\langle \text{VIEW-CHANGE}, v, P, j\rangle_{\sigma_j})_i$
2. $\text{VIEW-CHANGE}(v)_i$
3. $\text{RECEIVE}(\langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_j})_i$,
4. $\text{RECEIVE}(\langle \text{PREPARE}, v, n, d, j\rangle_{\sigma_j})_i$,

5. RECEIVE($\langle$NEW-VIEW$, v, V, O, N\rangle_{\sigma_j})_i$,
6. SEND-PRE-PREPARE$(m, v, n)_i$, or
7. SEND-NEW-VIEW$(v, V)_i$

Actions of the first type never violate the lemma because they only insert view-change messages signed by replicas other than R_i in in_i. The effects of actions of the second type ensure that when a view-change message $\langle$VIEW-CHANGE$, v, P, i\rangle_{\sigma_i}$ is inserted in in_i the following condition is true: (a) $\forall v' < v : (\textit{last-prepared}(m, v', n, i) \Leftrightarrow \textit{last-prepared}(m, v', n, \mathcal{P}))$. Condition (a) and Invariant A.1.2 (condition 9) imply condition 1 of the invariant.

For the other types of actions, assume there exists at least a view change message for v signed by R_i in in_i before one of the other types of actions executes (otherwise the lemma would be vacuously true) and pick any $m' = \langle$VIEW-CHANGE$, v, P, i\rangle_{\sigma_i} \in in_i$. The inductive hypothesis ensures that the following condition holds before the actions execute:

$\forall m, n, v' < v : (\textit{last-prepared-b}(m, v', n, i, v) \Leftrightarrow \textit{last-prepared}(m, v', n, \mathcal{P}))$

Therefore, it is sufficient to prove that the actions preserve this condition. The logical value of $\textit{last-prepared}(m, v', n, \mathcal{P}))$ does not change (for all m', m, n, v') because the view-change messages in in_i are immutable.

To prove that the value of $\textit{last-prepared-b}(m, v', n, i, v)$ is also preserved (for all m', m, n, v'), we will first prove the following invariant (b): For any reachable state in an execution of A, any non-faulty replica R_i, and any view-change message $m' = \langle$VIEW-CHANGE$, v, P, i\rangle_{\sigma_i}$, $m' \in in_i \Rightarrow view_i \geq v$.

The proof for (b) is by induction on the length of the execution. It is vacuously true in the base case. For the inductive step, the only actions that can violate (b) are actions that insert view-change messages signed by R_i in in_i or actions that make $view_i$ smaller. Since there are no actions that make $view_i$ smaller, these actions have the form VIEW-CHANGE$(v)_i$. The effects of actions of this form ensure the invariant is preserved by setting $view_i$ to the view number in the view-change message.

Given (b) it is easy to see that the other types of actions do not violate condition 1 of the lemma. They only insert pre-prepare or prepare messages in in_i whose view number is equal to $view_i$ after the action executes. Invariant (b) guarantees that $view_i$ is greater than or equal to the view number v of any view-change message in in_i. Therefore, these actions cannot change the value of $\textit{last-prepared-b}(m, v', n, i, v)$ for any m', m, n, v'.

Condition (2) of the lemma can only be violated by actions that insert new-view messages in in_i or remove pre-prepare messages from in_i. Since no action ever removes pre-prepare messages from in_i, the only actions that can violate condition (2) are: RECEIVE($\langle$NEW-VIEW$, v, V, O, N\rangle_{\sigma_j})_i$ and SEND-NEW-VIEW$(v, V)_i$. The first type of action preserves condition (2) because it inserts all the pre-prepares in $O \cup N$ in in_i whenever it inserts the argument new-view message in in_i. The second type of action preserves condition (2) in a similar way. □

Invariant A.1.9 *The following is true of any reachable state in an execution of A,*

$$\forall\, i \in \mathcal{R},\, m \in \mathcal{M},\, v,n \in \mathbf{N} : ((\neg \textit{faulty}_i \;\wedge\; \textit{n-faulty} \leq f \;\wedge$$
$$\exists\, R : (|R| > f \;\wedge\; \forall\, k \in R : (\neg \textit{faulty}_k \;\wedge\; \textit{prepared}(m,v,n,k))) \Rightarrow$$
$$\forall\, v' > v \in \mathbf{N},\, m' \in \mathcal{M} : (\langle \text{PRE-PREPARE}, v', n, m' \rangle_{\sigma_{\textit{primary}(v')}} \in \textit{in}_i \Rightarrow m' = m))$$

Proof: Rather than proving the invariant directly, we will prove the following condition is true:

$$\forall\, i \in \mathcal{R},\, m \in \mathcal{M},\, v,n \in \mathbf{N} : ((\neg \textit{faulty}_i \;\wedge\; \textit{n-faulty} \leq f \;\wedge$$
$$\exists\, R : (|R| > f \;\wedge\; \forall\, k \in R : (\neg \textit{faulty}_k \;\wedge\; \textit{prepared}(m,v,n,k))) \Rightarrow$$
$$\forall\, v' > v \in \mathbf{N},\, \langle \text{NEW-VIEW}, v', V, O, N \rangle_{\sigma_{\textit{primary}(v')}} \in \textit{in}_i :$$
$$(\langle \text{PRE-PREPARE}, v', n, m \rangle_{\sigma_{\textit{primary}(v')}} \in O))$$

Condition (a) implies the invariant. Invariant A.1.2 (condition 4) states that there is never a pre-prepare message in $\textit{in}_i$ for a view $v' > 0$ without a correct new-view message in $\textit{in}_i$ for the same view. But if there is a correct new-view message $\langle \text{NEW-VIEW}, v', V, O, N \rangle_{\sigma_{\textit{primary}(v')}} \in \textit{in}_i$ then Invariant A.1.8 (condition 2) implies that $(O \cup N) \subseteq \textit{in}_i$. This and condition (a) imply that there is a $\langle \text{PRE-PREPARE}, v', n, m \rangle_{\sigma_{\textit{primary}(v')}} \in \textit{in}_i$ and Invariant A.1.2 (conditions 1,2 and 8) implies that no different pre-prepare message for sequence number n and view v' is ever in $\textit{in}_i$.

The proof is by induction on the number of views between v and v'. For the base case, $v = v'$, condition (a) is vacuously true. For the inductive step, assume condition (a) holds for v'' such that $v < v'' < v'$. We will show that it also holds for v'. Assume there exists a new-view message $m_1 = \langle \text{NEW-VIEW}, v', V_1, O_1, N_1 \rangle_{\sigma_{\textit{primary}(v')}}$ in $\textit{in}_i$ (otherwise (a) is vacuously true.) From Invariant A.1.2 (condition 5), this message must verify $\textit{correct-new-view}(m_1, v')$. This implies that it must contain $2f+1$ correct view-change messages for view v' from replicas in some set R_1.

Assume that the following condition is true (b) $\exists R : (|R| > f \wedge \forall k \in R : (\textit{faulty}_k = \textit{false} \wedge \textit{prepared}(m,v,n,k) = \textit{true}))$ (otherwise (a) is vacuously true.) Since there are only $3f+1$ replicas, R and R_1 intersect in at least one replica and this replica is not faulty; call this replica k. Let k's view-change message in m_1 be $m_2 = \langle \text{VIEW-CHANGE}, v', P_2, k \rangle_{\sigma_k}$.

Invariant A.1.4 implies $\textit{last-prepared-b}(m, v, n, k, v+1)$ is true because k is non-faulty and $\textit{prepared}(m, v, n, k) = \mathit{true}$. Therefore, one of the following conditions is true:

1. $\textit{last-prepared-b}(m, v, n, k, v')$
2. $\exists v'', m' : (v < v'' < v' \wedge \textit{last-prepared-b}(m', v'', n, k, v'))$

Since condition (a) implies the invariant, the inductive hypothesis implies that $m = m'$ in the second case. Therefore, Invariants A.1.1 and A.1.8 imply that (c) $\exists v_2 \geq v : \textit{last-prepared}(m, v_2, n, P_2)$

Condition (c), Invariant A.1.5, and the fact that $\textit{correct-new-view}(m_1, v')$ is true imply that one of the following conditions is true:

1. $\textit{last-prepared}(m, v_2, n, \textit{merge-P}(V_1))$
2. $\exists v'', m' : (v_2 < v'' < v' \wedge \textit{last-prepared}(m', v'', n, \textit{merge-P}(V_1)))$

In case (1), (a) is obviously true. If case (2) holds, Invariants A.1.1 and A.1.2 (condition 7) imply that there exists at least one non-faulty replica j with $\langle \text{PRE-PREPARE}, v'', n, m' \rangle_{\sigma_{primary(v'')}} \in in_j$. Since condition (a) implies the invariant, the inductive hypothesis implies that $m = m'$ in the second case. ∎

Invariant A.1.10 *The following is true of any reachable state in an execution of A,*

$$\begin{array}{l}\forall\, n, v, v' \in \mathbf{N},\ m, m' \in \mathcal{M} : (\textit{n-faulty} \leq f \Rightarrow \\ \quad (\exists\, R \subseteq \mathcal{R} : (|R| > f \ \wedge \forall\, k \in R : (\neg \textit{faulty}_k \wedge \textit{prepared}(m, v, n, k))) \wedge \\ \quad \exists\, R' \subseteq \mathcal{R} : (|R'| > f \ \wedge \forall\, k \in R' : (\neg \textit{faulty}_k \wedge \textit{prepared}(m', v', n, k)))) \Rightarrow D(m) = D(m'))\end{array}$$

Proof: Assume without loss of generality that $v \leq v'$. For the case $v = v'$, the negation of this invariant implies that there exist two requests m and m' ($D(m') \neq D(m)$), a sequence number n, and two non-faulty replicas R_i, R_j, such that *prepared*$(m, v, n, i) =$ *true* and *prepared*$(m', v, n, j) =$ *true*; this contradicts Invariant A.1.4.

For $v > v'$, assume this invariant is false. The negation of the invariant and the definition of the prepared predicate imply:

$$\begin{array}{l}\exists\, n, v, v' \in \mathbf{N},\ m, m' \in \mathcal{M} : (v > v' \wedge \textit{n-faulty} \leq f \wedge \\ \quad (\exists\, R \subseteq \mathcal{R} : (|R| > f \ \wedge \forall\, k \in R : (\neg \textit{faulty}_k \wedge \textit{prepared}(m, v, n, k))) \wedge \\ \quad \exists\, i \in \mathcal{R} : (\neg \textit{faulty}_i \wedge \langle \text{PRE-PREPARE}, v', n, m' \rangle_{\sigma_{primary(v')}} \in in_i) \wedge D(m) \neq D(m'))\end{array}$$

But this contradicts Invariant A.1.9 as long as the probability that $m \neq m'$ while $D(m) = D(m')$ is negligible. ∎

Invariant A.1.11 *The following is true of any reachable state in an execution of A,*

$$\begin{array}{l}\forall\, i, j \in \mathcal{R},\ n, v, v' \in \mathbf{N},\ m, m' \in \mathcal{M} : ((\neg \textit{faulty}_i \wedge \neg \textit{faulty}_j \wedge \textit{n-faulty} \leq f) \Rightarrow \\ \quad (\textit{committed}(m, v, n, i) \wedge \textit{committed}(m', v', n, j) \Rightarrow D(m) = D(m')))\end{array}$$

Invariant A.1.12 *The following is true of any reachable state in an execution of A,*

$$\begin{array}{l}\forall\, i \in \mathcal{R},\ n, v, v' \in \mathbf{N},\ m, m' \in \mathcal{M} : ((\neg \textit{faulty}_i \wedge \textit{n-faulty} \leq f) \Rightarrow (\textit{committed}(m, v, n, i) \wedge \\ \quad \exists\, R' \subseteq \mathcal{R} : (|R'| > f \ \wedge \forall\, k \in R' : (\neg \textit{faulty}_k \wedge \textit{prepared}(m', v', n, k)))) \Rightarrow D(m) = D(m'))\end{array}$$

Proof: Both Invariant A.1.11 and A.1.12 are implied by Invariants A.1.10 and A.1.7. ∎

Rather than proving that A implements S directly, we will prove that A implements S', which replaces the value of the state machine in S by the history of all the operations executed. S' is better suited for the proof and we will use a simple simulation relation to prove that it implements S. We start by defining a set of auxiliary functions that will be useful in the proof.

Definition A.1.13 *We define the following functions inductively:*

$$
\begin{aligned}
&\mathit{val} \;:\; (\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^* \;\rightarrow\; \mathcal{S} \\
&\mathit{last\text{-}rep} \;:\; (\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^* \;\rightarrow\; (\,\mathcal{C} \rightarrow \mathcal{O}') \\
&\mathit{last\text{-}rep\text{-}t} \;:\; (\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^* \;\rightarrow\; (\,\mathcal{C} \rightarrow \mathbf{N})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{val}(\lambda) \;=\; s_o \\
&\forall\, c \;:\; (\mathit{last\text{-}rep}(\lambda)(c) \;=\; \mathit{null\text{-}rep}) \\
&\forall\, c \;:\; (\mathit{last\text{-}rep\text{-}t}(\lambda)(c) \;=\; 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{val}(\mu.\langle n, o, t, c\rangle) \;=\; s \\
&\mathit{last\text{-}rep}(\mu.\langle n, o, t, c\rangle)(c) \;=\; r \\
&\mathit{last\text{-}rep\text{-}t}(\mu.\langle n, o, t, c\rangle)(c) \;=\; t \\
&\forall\, c' \;\neq\; c \;:\; (\mathit{last\text{-}rep}(\mu.\langle n, o, t, c\rangle)(c') \;=\; \mathit{last\text{-}rep}(\mu)(c')) \\
&\forall\, c' \;\neq\; c \;:\; (\mathit{last\text{-}rep\text{-}t}(\mu.\langle n, o, t, c\rangle)(c') \;=\; \mathit{last\text{-}rep\text{-}t}(\mu)(c'))
\end{aligned}
$$

where $(r, s) \;=\; g(c, o, \mathit{val}(\mu))$

Automaton S' has the same signature as S except for the addition of an internal action EXECUTE-NULL. It also has the same state components except that the *val* component is replaced by a sequence of operations:

$\mathit{hist} \in (\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^*$, initially λ;

and there is a new *seqno* component:

$\mathit{seqno} \in \mathbf{N}$, initially 0.

Similarly to S, the transitions for S' are only defined when $\mathit{n\text{-}faulty} \leq f$. Also, the transitions for S' are identical to S's except for those defined bellow.

EXECUTE(o, t, c)
Pre: $\langle o, t, c\rangle \in \mathit{in}$
Eff: $\mathit{seqno} := \mathit{seqno} + 1$
$\quad \mathit{in} := \mathit{in} - \{\langle o, t, c\rangle\}$
$\quad$ if $t > \mathit{last\text{-}rep\text{-}t}(\mathit{hist})(c)$ then
$\quad\quad \mathit{hist} := \mathit{hist}.\langle \mathit{seqno}, o, t, c\rangle$
$\quad\quad \mathit{out} := \mathit{out} \cup \{\langle \mathit{last\text{-}rep}(c), t, c\rangle\}$

EXECUTE-NULL
Eff: $\mathit{seqno} := \mathit{seqno} + 1$

The EXECUTE-NULL actions allow the *seqno* component to be incremented without removing any tuple from *in*. This is useful to model execution of *null* requests.

Theorem A.1.14 *S' implements S*

Proof: The proof uses a *forward simulation* $\mathcal{F}$ from S' to S. $\mathcal{F}$ is defined as follows:

Definition A.1.15 *$\mathcal{F}$ is a subset of* states(S') × states(S)*; (x, y) is an element of $\mathcal{F}$ (also written as $y \in \mathcal{F}[x]$) if and only if all the following conditions are satisfied:*

1. All state components with the same name are equal in x and y.
2. $x.\mathit{val} = \mathit{val}(y.\mathit{hist})$
3. $x.\mathit{last\text{-}rep\text{-}t}_c = \mathit{last\text{-}rep}(y.\mathit{hist})(c), \forall c \in \mathcal{C}$

To prove that $\mathcal{F}$ is in fact a forward simulation from S' to S one most prove that both of the following are true [Lyn96].

1. For all $x \in \textit{start}(S')$, $\mathcal{F}[x] \cap \textit{start}(S) \neq \{\}$
2. For all $(x, \pi, x') \in \textit{trans}(S')$, where x is a reachable state of S', and for all $y \in \mathcal{F}[x]$, where y is reachable in S, there exists an execution fragment α of S starting with y and ending with some $y' \in \mathcal{F}[x']$ such that $\textit{trace}(\alpha) = \textit{trace}(\pi)$.

It is clear that $\mathcal{F}$ verifies the first condition because all variables with the same name in S and S' are initialized to the same values and, since *hist* is initially equal to λ, $x.\textit{val} = s_o = \textit{val}(\lambda)$ and $x.\textit{last-rep-t}_c = 0 = \textit{last-rep}(\lambda)(c)$.

We use case analysis to show that the second condition holds for each $\pi \in \textit{acts}(S')$. For all actions π except EXECUTE-NULL, let α consist of a single π step. For π = EXECUTE-NULL, let α be λ. It is clear that this satisfies the second condition for all actions but EXECUTE. For $\pi = \text{EXECUTE}(o, t, c)$, definition A.1.13 and the inductive hypothesis (i.e., $x.\textit{val} = \textit{val}(y.\textit{hist})$ and $x.\textit{last-rep-t}_c = \textit{last-rep}(y.\textit{hist})(c)$) ensure that $y' \in \mathcal{F}[x']$. □

Definition A.1.16 *We define the function prefix :* $(\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^* \rightarrow (\mathbf{N} \times \mathcal{O} \times \mathbf{N} \times \mathcal{C})^*$ *as follows: prefix*(μ, n) *is the subsequence obtained from* μ *by removing all tuples whose first component is greater than* n.

Invariant A.1.17 *The following is true of any reachable state in an execution of* S'*,*
$\forall \langle n, o, t, c\rangle \in \textit{hist} : (t > \textit{last-rep-t}(\textit{prefix}(\textit{hist}, n-1))(c))$

Proof: The proof is by induction on the length of the execution. The initial states of S' verify the condition vacuously because *hist* is initially λ. For the inductive step, the only actions that can violate the invariant are those that modify *hist*, i.e., EXECUTE(o, t, c). But these actions only modify *hist* if $t > \textit{last-rep-t}(\textit{hist})(c)$. □

Invariant A.1.18 *The following are true of any reachable state in an execution of* S'*:*

1. $\forall \langle n, o, t, c\rangle \in \textit{hist} : (\neg \textit{faulty}_c \Rightarrow t \leq \textit{last-req}_c)$
2. $\forall \langle o, t, c\rangle \in \textit{in} : (\neg \textit{faulty}_c \Rightarrow t \leq \textit{last-req}_c)$

Proof: The proof is by induction on the length of the execution. The initial states of S' verify the condition vacuously because *hist* is initially λ and *in* is empty. For the inductive step, since no action ever decrements $\textit{last-req}_c$ or changes $\textit{faulty}_c$ from true to false, the only actions that can violate the invariant are those that append tuples from a non-faulty client c to *hist*, i.e., EXECUTE(o, t, c) or to *in*, REQUEST(o, c). The EXECUTE actions only append a tuple $\langle n, o, t, c\rangle$ to *hist* if $\langle o, t, c\rangle \in \textit{in}$; therefore, the inductive hypothesis for condition 2 implies that they preserve the invariant. The REQUEST actions also preserve the invariant because the tuple $\langle o, t, c\rangle$ inserted in *in* has t equal to the value of $\textit{last-req}_c$ after the action executes. □

We are now ready to prove the main theorem in this section.

Theorem A.1.19 *A implements S*

Proof: We prove that A implements S', which implies that A implements S (Theorem A.1.14.) The proof uses a *forward simulation* $\mathcal{G}$ from A' to S' (A' is equal to A but with all output actions not in the external signature of S hidden.) $\mathcal{G}$ is defined as follows.

Definition A.1.20 *$\mathcal{G}$ is a subset of* $\text{states}(A') \times \text{states}(S')$*; (x, y) is an element of $\mathcal{G}$ if and only if the following are satisfied:*

1. $\forall i \in \mathcal{R} : (x.\mathit{faulty}_i = y.\mathit{faulty\text{-}replica}_i)$
2. $\forall c \in \mathcal{C} : (x.\mathit{faulty}_c = y.\mathit{faulty\text{-}client}_c)$

and the following are satisfied when n-faulty $\leq f$

3. $\forall c \in \mathcal{C} : (\neg x.\mathit{faulty}_c \Rightarrow x.\mathit{last\text{-}req}_c = y.\mathit{last\text{-}req}_c)$
4. $\forall i \in \mathcal{R} : (\neg x.\mathit{faulty}_i \Rightarrow x.\mathit{last\text{-}exec}_i \leq y.\mathit{seqno})$
5. $\forall i \in \mathcal{R} : (\neg x.\mathit{faulty}_i \Rightarrow x.\mathit{val}_i = \mathit{val}(\mathit{prefix}(y.\mathit{hist}, x.\mathit{last\text{-}exec}_i)))$
6. $\forall i \in \mathcal{R} : (\neg x.\mathit{faulty}_i \Rightarrow \forall c \in \mathcal{C} : (x.\mathit{last\text{-}rep}_i(c) = \mathit{last\text{-}rep}(\mathit{prefix}(y.\mathit{hist}, x.\mathit{last\text{-}exec}_i))(c)))$
7. $\forall i \in \mathcal{R} : (\neg x.\mathit{faulty}_i \Rightarrow \forall c \in \mathcal{C} : (x.\mathit{last\text{-}rep\text{-}t}_i(c) = \mathit{last\text{-}rep\text{-}t}(\mathit{prefix}(x.\mathit{hist}, y.\mathit{last\text{-}exec}_i))(c)))$
8. $\forall 0 < n \leq y.\mathit{seqno}$:
   $\exists \langle n, o, t, c \rangle \in y.\mathit{hist} : (\exists R \subseteq \mathcal{R}, v \in \mathbf{N} : (|R| > 2f - y.\mathit{n\text{-}faulty} \wedge$
   $\forall k \in R : (\neg x.\mathit{faulty}_k \wedge \mathit{prepared}(\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}, v, n, A'.k)))) \vee$
   $(\neg \exists \langle n, o, t, c \rangle \in y.\mathit{hist} \wedge$
   $(\exists R \subseteq \mathcal{R}, v, t \in \mathbf{N}, o \in \mathcal{O}, c \in \mathcal{C} : (|R| > 2f - y.\mathit{n\text{-}faulty} \wedge t \leq \mathit{last\text{-}rep\text{-}t}(\mathit{prefix}(y.\mathit{hist}, n-1))(c)) \wedge$
   $\forall k \in R : (\neg x.\mathit{faulty}_k \wedge \mathit{prepared}(\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}, v, n, A'.k))))$
   $\vee\ \exists R \subseteq \mathcal{R}, v \in \mathbf{N} : (|R| > 2f - y.\mathit{n\text{-}faulty} \wedge \forall k \in R : (\neg x.\mathit{faulty}_k \wedge \mathit{prepared}(\mathit{null}, v, n, A'.k)))))$
9. $\forall \langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i} \in (x.\mathit{out}_i \cup \{m | \exists X : (m, X) \in x.\mathit{wire}\} \cup x.\mathit{in}_c)$:
   $(\neg x.\mathit{faulty}_i \Rightarrow \exists \langle n, o, t, c \rangle \in y.\mathit{hist} : (r = \mathit{last\text{-}rep}(\mathit{prefix}(y.\mathit{hist}, n))(c)))$
10. $\forall \langle n, o, y.\mathit{last\text{-}req}_c, c \rangle \in y.\mathit{hist}$:
    $((\neg x.\mathit{faulty}_c \wedge x.\mathit{out}_c \neq \{\}) \Rightarrow \exists \langle \mathit{last\text{-}rep}(\mathit{prefix}(y.\mathit{hist}, n))(c), y.\mathit{last\text{-}req}_c, c \rangle \in y.\mathit{out})$
11. Let $M_c = x.\mathit{out}_c \cup \{m | \exists i \in \mathcal{R} : (\neg x.\mathit{faulty}_i \wedge m \in x.\mathit{in}_i \cup x.\mathit{out}_i\} \cup \{m | \exists X : (m, X) \in x.\mathit{wire}\}$,
    and $M_c^1 = \mathit{merge\text{-}P}(\{m = \langle \text{VIEW-CHANGE}, v, P, j \rangle_{\sigma_j} | m \in M_c \vee$
    $\exists \langle \text{NEW-VIEW}, v, V, O, N \rangle_{\sigma_j} \in M_c : (m \in V)\})$,
    $\forall c \in \mathcal{C} : (\neg x.\mathit{faulty}_c \Rightarrow \forall o \in \mathcal{O}, t \in \mathbf{N} : ((m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c} \in M_c \vee$
    $\exists \langle \text{PRE-PREPARE}, v, n, m \rangle_{\sigma_j} \in M_c \cup M_c^1) \Rightarrow (\langle o, t, c \rangle \in y.\mathit{in} \vee \exists n : (\langle n, o, t, c \rangle \in y.\mathit{hist}))))$

The intuition behind the definition of $\mathcal{G}$ is the following. The first two conditions say that the same replicas and clients are faulty in related A' and S' states. The next condition requires the last request timestamp for all non-faulty clients to be equal in related states. Condition 4 says that automaton A' cannot execute requests with sequence numbers that have not yet been executed in S'. Conditions 5 to 7 state that $x.\mathit{val}_i$, $x.\mathit{last\text{-}rep}_i$, and $x.\mathit{last\text{-}rep\text{-}t}_i$ can be obtained by executing the prefix of y's history up to the sequence number of the last request executed by replica i in x.

Condition 8 is the most interesting because it relates the *commit point* for requests in A' with the execution of regular and null requests in S'. All sequence numbers in y that correspond to a request in y's history must be prepared by at least $f + 1$ correct replicas in x. The other sequence numbers must correspond to a request with an old timestamp or a null request that is prepared by at least $f + 1$ correct replicas in x. Condition 9 says that replies from non-faulty replicas in A' must correspond to replies returned in S'. The next condition requires every request from a correct client in y's history to have a reply in $y.out$ if that reply was not received by the client in x. The final condition states that all requests in x must be either in y's history or in $y.in$.

Note that most of the conditions in the definition of $\mathcal{G}$ only need to hold when *n-faulty* $\leq f$, for *n-faulty* $> f$ any relation will do because the behavior of S' is unspecified.

To prove that $\mathcal{G}$ is in fact a forward simulation from A' to S' one most prove that both of the following are true.

1. For all $x \in \textit{start}(A')$, $\mathcal{G}[x] \cap \textit{start}(S') \neq \{\}$
2. For all $(x, \pi, x') \in \textit{trans}(A')$, where x is a reachable state of A', and for all $y \in \mathcal{G}[x]$, where y is reachable in S', there exists an execution fragment α of S' starting with y and ending with some $y' \in \mathcal{G}[x']$ such that $\textit{trace}(\alpha) = \textit{trace}(\pi)$.

It is easy to see that the first condition holds. We use case analysis to show that the second condition 2 holds for each $\pi \in \textit{acts}(A')$

Non-faulty proxy actions. If $\pi = \text{REQUEST}(o)_c$, $\pi = \text{CLIENT-FAILURE}_c$, or $\pi = \text{REPLY}(r)_c$, let α consist of a single π step. $\mathcal{G}$ is preserved in a trivial way if π is a CLIENT-FAILURE action. If π is a REQUEST action, neither π nor α modify the variables involved in all conditions in the definition of $\mathcal{G}$ except 3, and 10 and 11. Condition 3 is preserved because both π and α increment $y.\textit{last-req}_c$. Condition 10 is also preserved because Invariant A.1.18 implies that there are no tuples in $y.\textit{hist}$ with timestamp $y'.\textit{last-req}_c$ and α does not add any tuple to $y.\textit{hist}$. Even though π inserts a new request in $x.\textit{out}_c$, condition 11 is preserved because α inserts $\langle o, t, c\rangle$ in $y.\textit{in}$.

If π is a $\text{REPLY}(r)_c$ action that is enabled in x, the $\text{REPLY}(r)_c$ action in α is also enabled. Since there are less than f faulty replicas, the precondition of π ensures that there is at least one non-faulty replica i and a view v such that $\langle \text{REPLY}, v, x.\textit{last-req}_c, c, i, r\rangle_{\sigma_i} \in x.\textit{in}_c$ and that $x.\textit{out}_c \neq \{\}$. Therefore, the inductive hypothesis (conditions 9 and 10) implies that $\langle r, t, c\rangle \in y.\textit{out}$ and thus $\text{REPLY}(r)_c$ is enabled. $\mathcal{G}$ is preserved because π ensures that $x'.\textit{out}_c = \{\}$.

If $\pi = \text{RECEIVE}(m)_c$, or $\pi = \text{SEND}(m, X)_c$, let α be λ. This preserves $\mathcal{G}$ because $y \in \mathcal{G}[x]$ and the preconditions require that the reply message being received is in some tuple in $x.\textit{wire}$ and the request message being sent is in $x.\textit{out}_c$.

Internal channel actions. If π is a $\text{MISBEHAVE}(m, X, X')$ action, let α be λ. $\mathcal{G}$ is preserved because π does not add new messages to $x.\textit{wire}$ and retains a tuple with m on $x'.\textit{wire}$.

Non-faulty replica actions. For all actions π except $\pi = \text{REPLICA-FAILURE}_i$ and $\pi = \text{EXECUTE}(m, v, n)_i$, let α be λ. It is clear that this could only violate conditions 8, 9 and 11 because these actions do not modify the state components involved in the other conditions. They can not violate condition 8; since no messages are ever removed from $\textit{in}_k$ (where k is any non-faulty replica), if *prepared*$(m, v, n, k) = $ *true*, it remains true for the entire execution or until replica k becomes faulty. And these actions do not violate conditions 9 and 11 because any request or reply messages they add to $x.in_i$, $x.out_i$, or $x.wire$ (either directly or as part of other messages) was already in $x.wire$, $x.in_i$, or $x.out_i$.

For $\pi = $ REPLICA-FAILURE$_i$, let α consist of a single π step. This does not violate the conditions in the definition of $\mathcal{G}$. For conditions other than 1 and 8, it either does not change variables involved in these conditions (2 and 3), or makes them vacuously true. Condition 1 is satisfied in a trivial way because α also sets $y.\textit{faulty-replica}_i$ to true. And condition 8 is not violated because the size of the sets R in the condition is allowed to decrease when additional replicas become faulty.

Non-faulty replica execute (non-null request.)

For $\pi = $ EXECUTE$(\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}, v, n)_i$, there are two cases: if $x.\textit{last-exec}_i < y.\textit{seqno}$, let α be λ; otherwise, let α consist of the execution of a single EXECUTE(o, t, c) action preceded by FAULTY-REQUEST(o, t, c) in the case where $x.\textit{faulty}_c = $ *true*. In any of these cases, it is clear that only conditions 4 to 11 can be violated.

For the case where $\alpha = \lambda$, conditions 4, 8, 10 and 11 are also preserved in a trivial way. For the other conditions we consider two cases (a) $t >$ *last-rep-t*$_i(c)$ and (b) otherwise. The precondition of π ensures that $x.\textit{committed}(\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}, v, n, i)$ is true. In case (a), this precondition, Invariant A.1.12, and the definition of $\mathcal{G}$ (condition 8) imply that there is a tuple in $y.\textit{hist}$ with sequence number n and that it is equal to $\langle n, o, t, c\rangle$. Therefore, conditions 5 to 7 and 9 are preserved. In case (b), the precondition of π, Invariant A.1.12, the definition of $\mathcal{G}$ (condition 8), and Invariant A.1.17 imply that there is no tuple with sequence number n in $y.\textit{hist}$. Therefore, conditions 5 to 9 are preserved in this case.

For the case where $\alpha \neq \lambda$, when π is enabled in x the actions in α are also enabled in y. In the case where c is faulty, FAULTY-REQUEST(o, t, c) is enabled and its execution enables EXECUTE(o, t, c). Otherwise, since $y \in \mathcal{G}[x]$, condition 11 in Definition A.1.20 and the precondition of π imply that EXECUTE(o, t, c) is enabled in y.

It is easy to see that conditions 4 to 7 and 9 to 11 are preserved. For condition 8, we consider two cases (a) $t >$ *last-rep-t*$_i(c)$ and (b) otherwise. In both cases, the precondition of π ensures that $x.\textit{committed}(\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}, v, n, i)$ is true. This precondition, Invariant A.1.7 and the fact that α appends a tuple $\langle y'.\textit{seqno}, o, t, c\rangle$ to $y.\textit{hist}$, ensure that condition 8 is preserved in this case. In case (b), the precondition Invariant A.1.7 and the assumption that $t \leq$ *last-rep-t*$_i(c)$, ensure that condition 8 is preserved also in this case.

Non-faulty replica execute (null request.)

For $\pi = $ EXECUTE$(null, v, n)_i$, if $x.\textit{last-exec}_i < y.\textit{seqno}$, let α be λ; otherwise, let α consist of the execution of a single EXECUTE-NULL action. Execution of a *null* request only increments $x.\textit{last-exec}_i$ and α can at most increment $y.\textit{seqno}$. Therefore, only conditions 4 to 8 can be violated. Condition 4 is not violated because α increments $y.\textit{seqno}$ in the case where $x.\textit{last-exec}_i = y.\textit{seqno}$.

For the case where, $\alpha = \lambda$, conditions 5 to 7 are also not violated because α does not append any new tuple to $y.hist$ and all tuples in $y.hist$ have sequence number less than $y'.seqno$; therefore, $prefix(y.hist, x.last\text{-}exec_i) = prefix(y'.hist, x'.last\text{-}exec_i)$. Since the precondition of π implies that $x.committed(null, v, n, i)$ is true, Invariant A.1.7 ensures condition 8 is also preserved in this case.

For the case where α consists of a EXECUTE-NULL step, $x.committed(null, v, n, i)$, $n\text{-}faulty \leq f$, Invariant A.1.12, and the definition of $\mathcal{G}$ (condition 8) imply that there is no tuple in $y'.hist$ with sequence number $x'.last\text{-}exec_i$; therefore, $prefix(y.hist, x.last\text{-}exec_i) = prefix(y'.hist, x'.last\text{-}exec_i)$.

Faulty replica actions. If π is an action of a faulty replica i (i.e., $x.faulty_i = true$), let α be λ. Since π can not modify $faulty_i$ and a faulty replica cannot forge the signature of a non-faulty automaton this preserves $\mathcal{G}$ in a trivial way.

Faulty proxy actions. If π is an action of a faulty proxy c (i.e., $x.faulty_c = true$), let α consist of a single π step for REQUEST, REPLY and CLIENT-FAILURE actions and λ for the other actions. Since π can not modify $faulty_c$ and faulty clients cannot forge signatures of non-faulty automata this preserves $\mathcal{G}$ in a trivial way. Additionally, if π is a REPLY action enabled in x, π is also enabled in y. □

A.2 Algorithm With Garbage Collection

We are now ready to prove that A_{gc} (the algorithm specified in Section 2.4) implements S. We start by introducing some definitions and proving a couple of invariants. Then, we use a simulation relation to prove A_{gc} implements A.

Definition A.2.1 *We define the following functions inductively:*

Let $\mathcal{RM} = \{\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c} \mid o \in \mathcal{O} \wedge t \in \mathbf{N} \wedge c \in \mathcal{C}\} \cup \{null\}$,
$r\text{-}val : \mathcal{RM}^* \rightarrow \mathcal{S}$
$r\text{-}last\text{-}rep : \mathcal{RM}^* \rightarrow (\mathcal{C} \rightarrow \mathcal{O}')$
$r\text{-}last\text{-}rep\text{-}t : \mathcal{RM}^* \rightarrow (\mathcal{C} \rightarrow \mathbf{N})$

$r\text{-}val(\lambda) = s_o$
$\forall c \in \mathcal{C} : (r\text{-}last\text{-}rep(\lambda)(c) = null\text{-}rep)$
$\forall c \in \mathcal{C} : (r\text{-}last\text{-}rep\text{-}t(\lambda)(c) = 0)$

$\forall \mu \in \mathcal{RM}^+$,
 $r\text{-}val(\mu.null) = r\text{-}val(\mu)$
 $r\text{-}last\text{-}rep(\mu.null) = r\text{-}last\text{-}rep(\mu)$
 $r\text{-}last\text{-}rep\text{-}t(\mu.null) = r\text{-}last\text{-}rep\text{-}t(\mu)$

$\forall \langle \text{REQUEST}, o, t, c\rangle_{\sigma_c} \in \mathcal{RM},\ \mu \in \mathcal{RM}^+$,
 $\forall c' \neq c : (r\text{-}last\text{-}rep(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c') = r\text{-}last\text{-}rep(\mu)(c'))$
 $\forall c' \neq c : (r\text{-}last\text{-}rep\text{-}t(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c') = r\text{-}last\text{-}rep\text{-}t(\mu)(c'))$
 if $t > r\text{-}last\text{-}rep\text{-}t(\mu)(c)$ then
  let $(r, s) = g(c, o, r\text{-}val(\mu))$
   $r\text{-}val(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}) = s$
   $r\text{-}last\text{-}rep(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c) = r$
   $r\text{-}last\text{-}rep\text{-}t(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c) = t$

else
$r\text{-}val(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}) = r\text{-}val(\mu)$
$r\text{-}last\text{-}rep(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c) = r\text{-}last\text{-}rep(\mu)(c)$
$r\text{-}last\text{-}rep\text{-}t(\mu.\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c})(c) = r\text{-}last\text{-}rep\text{-}t(\mu)(c)$

Definition A.2.2 *We define the following subsets of $\mathcal{M}$ and predicate:*

$Wire \equiv \{ m \mid \exists X : ((m, X) \in wire) \}$
$Wire{+}o \equiv Wire \cup \{ m \mid \exists j \in \mathcal{R} : (\neg faulty_j \wedge m \in out_j) \}$
$Wire{+}io \equiv Wire{+}o \cup \{ m \mid \exists j \in \mathcal{R} : (\neg faulty_j \wedge m \in in_j) \}$

$$
\begin{aligned}
&\text{committed-Wire}(s, l, t, n, v, \mu) \equiv \\
&\quad \exists\, m_1 ... m_n = \mu \in \mathcal{RM}^* : (s = \text{r-val}(\mu) \wedge l = \text{r-last-rep}(\mu) \wedge t = \text{r-last-rep-t}(\mu) \wedge \\
&\quad\ \forall\, 0 < k \le n : (\exists\, v' \le v,\ R : (|R| > 2f \wedge \\
&\qquad\qquad\qquad \forall\, q \in R : (\langle \text{COMMIT}, v', k, D(m_k), q\rangle_{\sigma_q} \in \text{Wire+o})) \\
&\qquad\qquad\quad \wedge (\exists\, v' \le v : (\langle \text{PRE-PREPARE}, v', k, m_k\rangle_{\sigma_{\text{primary}(v')}} \in \text{Wire+o}) \\
&\qquad\qquad\qquad \vee\, m_k \in \text{Wire+o})))
\end{aligned}
$$

The functions in Definition A.2.1 compute the value of the various checkpoint components after executing a sequence of requests. The predicate *committed-Wire* relates the value of the checkpoint components with a sequence of committed requests in *Wire+o* that can be executed to obtain those values (where *Wire+o* is the set of messages in the multicast channel or in the *out* variables of correct replicas). The following invariant states that *committed-Wire* is true for the state components of correct replicas and the checkpoint messages they send.

Invariant A.2.3 *The following is true of any reachable state in an execution of A_{gc}:*

1. $\forall\, i \in \mathcal{R} : ((\neg faulty_i \wedge n\text{-}faulty \le f) \Rightarrow$
   $\exists\, \mu \in \mathcal{RM}^* : committed\text{-}Wire(val_i, last\text{-}rep_i, last\text{-}rep\text{-}t_i, last\text{-}exec_i, view_i, \mu))$

2. $\forall\, i \in \mathcal{R} : (\neg faulty_i \wedge n\text{-}faulty \le f) \Rightarrow$
   $\forall\, \langle \text{CHECKPOINT}, v, n, D(\langle s, l, t\rangle), i\rangle_{\sigma_i} \in N : (\exists\, \mu \in \mathcal{RM}^* : committed\text{-}Wire(s, l, t, n, v, \mu))$

where:
$N = \{ m \mid m \in Wire{+}io \vee \exists\, \langle \text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j} \in Wire{+}io : (m \in C) \vee$
$\quad \exists\, \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j} \in Wire{+}io : (\exists\, \langle \text{VIEW-CHANGE}, v, n, s, C, P, q\rangle_{\sigma_q} \in V : (m \in C)) \}$,

Proof: The proof is by induction on the length of the execution. For the base case, the initializations ensure that $val_i = r\text{-}val(\lambda)$, $last\text{-}rep_i = r\text{-}last\text{-}rep(\lambda)$, and $last\text{-}rep\text{-}t_i = r\text{-}last\text{-}rep\text{-}t(\lambda)$. Therefore, 1 is obviously true in the base case and 2 is also true because all the checkpoint messages $\langle \text{CHECKPOINT}, v, n, D(\langle s, l, t\rangle), i\rangle_{\sigma_i} \in N$ have $s = val_i, l = last\text{-}rep_i, t = last\text{-}rep\text{-}t_i$.

For the inductive step, assume that the invariant holds for every state of any execution α of length at most l. We will show that the lemma also holds for any one step extension α_1 of α. The only actions that can violate 1 are actions that change $val_i, last\text{-}rep_i, last\text{-}rep\text{-}t_i, last\text{-}exec_i$, decrement $view_i$, or remove messages from *Wire+o*. But no actions ever decrement $view_i$. Similarly, no actions ever remove messages from *Wire+o* because *wire* remembers all messages that were ever sent over the multicast channel and messages are only removed from out_j (for any non-faulty replica j) when they are sent over the multicast channel. Therefore, the only actions that can violate 1 are:

1. RECEIVE($\langle$NEW-VIEW$, v, V, O, N\rangle_{\sigma_j})_i$
2. EXECUTE$(m, v, n)_i$
3. SEND-NEW-VIEW$(v, V)_i$

The inductive hypothesis of condition 2 ensures that actions of the first and third type do not violate condition 1 because they set val_i, $last\text{-}rep_i$, $last\text{-}rep\text{-}t_i$ and $last\text{-}exec_i$ to the corresponding values in a checkpoint message from a non-faulty replica.

Actions of the second type also do not violate 1 because of the inductive hypothesis, and because the executed request, m_n, verifies $committed(m_n, v, n, i)$ for $v \leq view_i$ and $n = last\text{-}exec_i + 1$. Since $committed(m_n, v, n, i)$ is true, the $2f + 1$ commits and the pre-prepare (or m_n) necessary for *committed-Wire* to hold are in in_i. These messages were either received by i over the multicast channel or they are messages from i, in which case they are in out_i or have already been sent over the multicast channel.

The only actions that can violate condition 2 are those that insert checkpoint messages in N:

1. RECEIVE($\langle$CHECKPOINT$, v, n, d, i\rangle_{\sigma_i})_j$
2. RECEIVE($\langle$VIEW-CHANGE$, v, n, s, C, P, q\rangle_{\sigma_q})_j$
3. RECEIVE($\langle$NEW-VIEW$, v, V, O, N\rangle_{\sigma_q})_j$
4. SEND$(m, R)_i$
5. EXECUTE$(m, v, n)_j$
6. SEND-VIEW-CHANGE$(v)_j$
7. SEND-NEW-VIEW$(v, V)_j$

where j is any non-faulty replica. Actions of types 1, 2, 4, and 6 preserve 2 because the checkpoints they insert into N are already in N before the action executes and because of the inductive hypothesis. Actions of types 3 and 7 may insert a new checkpoint message from j into N; but they also preserve condition 2 because this message has the same sequence number and checkpoint digest as some checkpoint message from a non-faulty replica that is already in N before the action executes and because of the inductive hypothesis. Finally, the argument to show that actions of the fifth type preserve 1 also shows that they preserve condition 2. $\square$

Invariant A.2.4 *The following is true of any reachable state in an execution of A:*

$$n\text{-}faulty \leq f \Rightarrow \forall \mu, \mu' \in \mathcal{RM}^* : ((\exists s, l, t, v, s', l', t', v' : (committed\text{-}Wire(s, l, t, n, v, \mu) \land committed\text{-}Wire(s', l', t', n', v', \mu')) \land \mu.length \leq \mu'.length) \Rightarrow \exists \mu'' \in \mathcal{RM}^* : (\mu' = \mu.\mu''))$$

Proof: (By contradiction) Suppose that the invariant is false. Then, there may exist some sequence number k ($0 < k \leq \mu.length$) and two different requests m_{k_1} and m_{k_2} such that:

$\exists\, v_1,\ R_1\ :\ (|R_1|\ >\ 2f\ \wedge\ \forall\, q\ \in\ R_1\ :\ (\langle \text{COMMIT}, v_1, k, D(m_{k_1}), q\rangle_{\sigma_q}\ \in\ \mathit{Wire}{+}o))$ and
$\exists\, v_2,\ R_2\ :\ (|R_2|\ >\ 2f\ \wedge\ \forall\, q\ \in\ R_2\ :\ (\langle \text{COMMIT}, v_2, k, D(m_{k_2}), q\rangle_{\sigma_q}\ \in\ \mathit{Wire}{+}o))$

This, Invariant A.1.1 and Invariant A.1.6 contradict Invariant A.1.10. □

Invariant A.2.4 states that if *committed-Wire* is true for two sequences of messages in A (which is the algorithm without garbage collection) then one sequence must be a prefix of the other. Now we can prove our main result: A_{gc} implements S.

Theorem A.2.5 *A_{gc} implements S*

Proof: We prove that A_{gc} implements A, which implies that it implements S (Theorems A.1.19 and A.1.14.) The proof uses a forward simulation $\mathcal{H}$ from A'_{gc} to A' (A'_{gc} is equal to A_{gc} but with all output actions not in the external signature of S hidden.)

Definition A.2.6 *$\mathcal{H}$ is a subset of* states(A'_{gc}) × states(A')*; (x, y) is an element of $\mathcal{H}$ if and only if all the following conditions are satisfied for any replica i such that x.*faulty$_i$ = false*, and for any replica j:*

1. The values of the state variables in y are equal to the corresponding values in x except for $y.\mathit{wire}$, $y.\mathit{in}_i$ and $y.\mathit{out}_i$.
2. $$\begin{aligned} & y.\mathit{in}_i\ -\ \{m\ =\ \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j}\ \vee\ m\ =\ \langle \text{PREPARE}, v, n, d, j\rangle_{\sigma_j}\ \vee \\ & \qquad m\ =\ \langle \text{COMMIT}, v, n, d, j\rangle_{\sigma_j}\mid m\ \in\ y.\mathit{in}_i\ \wedge\ n\ \leq\ x.h_i\} \\ & \quad -\{m \mid m\ \in\ y.\mathit{in}_i\ \wedge\ (\mathit{tag}(m, \text{VIEW-CHANGE})\ \vee\ \mathit{tag}(m, \text{NEW-VIEW}))\} \\ & =\ x.\mathit{in}_i\ -\ \{m\ =\ \langle \text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j}\ \vee\ m\ =\ \langle \text{PREPARE}, v, n, d, j\rangle_{\sigma_j}\ \vee \\ & \qquad m\ =\ \langle \text{COMMIT}, v, n, d, j\rangle_{\sigma_j}\mid m\ \in\ x.\mathit{in}_i\ \wedge\ n\ \leq\ x.h_i\} \\ & \quad -\{m \mid m\ \in\ x.\mathit{in}_i\ \wedge\ (\mathit{tag}(m, \text{CHECKPOINT})\ \vee\ \mathit{tag}(m, \text{VIEW-CHANGE})\ \vee\ \mathit{tag}(m, \text{NEW-VIEW}))\} \end{aligned}$$
3. Let $\mathit{consistent\text{-}vc}(m^1, m^2)\ \equiv$
   $$\begin{aligned} & \exists\, v, n, s, l, t, C, P, P', j\ :\ (m^1\ =\ \langle \text{VIEW-CHANGE}, v, n, \langle s, l, t\rangle, C, P, j\rangle_{\sigma_j}\ \wedge \\ & \quad m^2\ =\ \langle \text{VIEW-CHANGE}, v, P', j\rangle_{\sigma_j}\ \wedge \\ & \quad A'_{gc}.\mathit{correct\text{-}view\text{-}change}(m^1, v, j)\ \Leftrightarrow\ (A'.\mathit{correct\text{-}view\text{-}change}(m^2, v, j)\wedge \\ & \quad P\ =\ P' - \{m\ =\ \langle \text{PRE-PREPARE}, v', n', m'\rangle_{\sigma_k} \vee m\ =\ \langle \text{PREPARE}, v', n', d', k\rangle_{\sigma_k} \mid m\ \in\ P' \wedge n'\ \leq\ n\}))) \end{aligned}$$
   $$\begin{aligned} & \mathit{consistent\text{-}vc\text{-}set}(M^1, M^2)\ \equiv \\ & \quad \forall\, m^1\ \in\ M^1\ :\ (\exists\, m^2\ \in\ M^2\ :\ \mathit{consistent\text{-}vc}(m^1, m^2))\wedge \\ & \quad \forall\, m^2\ \in\ M^2\ :\ (\exists\, m^1\ \in\ M^1\ :\ \mathit{consistent\text{-}vc}(m^1, m^2)), \end{aligned}$$
   and let $y.\mathit{vc}_i\ =\ \{\langle \text{VIEW-CHANGE}, v, P, j\rangle_{\sigma_j}\ \in\ y.\mathit{in}_i\ \}$,
   $x.\mathit{vc}_i\ =\ \{\langle \text{VIEW-CHANGE}, v, n, \langle s, l, t\rangle, C, P, j\rangle_{\sigma_j}\ \in\ x.\mathit{in}_i\}$
   then $\mathit{consistent\text{-}vc\text{-}set}(x.\mathit{vc}_i, y.\mathit{vc}_i)$ is true
4. Let $\mathit{consistent\text{-}nv\text{-}set}(M_1, M_2)\ \equiv$
   $$\begin{aligned} & M_2\ =\ \{m^2 = \langle \text{NEW-VIEW}, v, V', O', N'\rangle_{\sigma_j} \mid \\ & \quad \exists\, m^1 = \langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j}\ \in\ M_1\ :\ (\mathit{consistent\text{-}vc\text{-}set}(V, V')\wedge \\ & \quad A'_{gc}.\mathit{correct\text{-}new\text{-}view}(m^1, v)\ \Leftrightarrow\ (A'.\mathit{correct\text{-}new\text{-}view}(m^2, v)\wedge \\ & \quad O\ =\ O'\ -\ \{m\ =\ \langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_j} \mid m\ \in\ O'\ \wedge\ n\ \leq\ \mathit{max\text{-}n}(V)\}\wedge \\ & \quad N\ =\ N'\ -\ \{m\ =\ \langle \text{PRE-PREPARE}, v, n, m'\rangle_{\sigma_j} \mid m\ \in\ N'\ \wedge\ n\ \leq\ \mathit{max\text{-}n}(V)\}))\}, \end{aligned}$$
   and let $y.\mathit{nv}_i\ =\ \{\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j}\ \in\ y.\mathit{in}_i\ \}$,
   $x.\mathit{nv}_i\ =\ \{\langle \text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j}\ \in\ x.\mathit{in}_i\ \}$
   then $\mathit{consistent\text{-}nv\text{-}set}(x.\mathit{nv}_i, y.\mathit{nv}_i)$ is true.
5. Let $\mathit{consistent\text{-}all}(M^1, M^2)\ \equiv$
   $$\begin{aligned} & \forall\, m\ \in\ M^1\ :\ (\exists\, m'\ \in\ M^2\ :\ (\mathit{tag}(m, \text{VIEW-CHANGE})\ \wedge\ \mathit{consistent\text{-}vc}(m, m'))\vee \\ & \quad (\mathit{tag}(m, \text{NEW-VIEW})\ \wedge\ \mathit{consistent\text{-}nv\text{-}set}(\{m\}, \{m'\}))\vee \\ & \quad (\neg \mathit{tag}(m, \text{VIEW-CHANGE})\ \wedge\ \neg \mathit{tag}(m, \text{NEW-VIEW})\ \wedge\ m\ =\ m')), \end{aligned}$$
   $X_i\ =\ x.\mathit{out}_i\ \cup\ \{\langle m\rangle_{\sigma_i} \mid \langle m\rangle_{\sigma_i}\ \in\ x.\mathit{Wire}\}\ -\{m \mid \mathit{tag}(m, \text{CHECKPOINT})\}$,
   and $Y_i\ =\ y.\mathit{out}_i\ \cup\ \{\langle m\rangle_{\sigma_i} \mid \langle m\rangle_{\sigma_i}\ \in\ y.\mathit{Wire}\}$,
   then $\mathit{consistent\text{-}all}(X_i. Y_i)$

6. Let $X_{\mathit{faulty}} = \{ \langle m \rangle_{\sigma_j} \mid x.\mathit{faulty}_j \wedge \langle m \rangle_{\sigma_j} \in x.\mathit{Wire}\}$,
$Y_{\mathit{faulty}} = \{ \langle m \rangle_{\sigma_j} \mid y.\mathit{faulty}_j \wedge \langle m \rangle_{\sigma_j} \in y.\mathit{Wire}\}$,
$\mathit{consistent\text{-}all}(X_{\mathit{faulty}}, Y_{\mathit{faulty}})$

7. $\forall \langle r \rangle_{\sigma_c} \in x.\mathit{Wire} : (\exists \langle r \rangle_{\sigma_c} \in y.\mathit{Wire})$

Additionally, we assume faulty automata in x are also faulty and identical in $\mathcal{H}[x]$ (i.e., they have the same actions and the same state.) Note that the conditions in the definition of $\mathcal{H}$ only need to hold when n-faulty $\leq f$, *for* n-faulty $> f$ *the behavior of S is unspecified.*

States related by $\mathcal{H}$ have the same values for variables with the same name with the exception of *wire*, and the *in* and *out* variables of non-faulty replicas. The second condition says that the *in* variables of non-faulty replicas have the same messages in related states with the exception of those messages that were garbage collected in x and view-change, new-view, and checkpoint messages.

Conditions 3 and 4 specify that view-change and new-view messages in $x.in_i$ and $y.in_i$ are *consistent*. These conditions define the notion of consistency precisely but the intuition is the following. A view-change message m in x is consistent with a view-change message m' in y if m contains exactly the pre-prepare and prepare messages in m' with sequence number greater than the checkpoint in m. Similarly, new-view messages are consistent if they contain consistent view-change messages and they propagate the same pre-prepares for the new-view with sequence number greater than the checkpoint that is propagated to the new view in A'_{gc}.

Condition 5 says that messages in the wire or *out* variables of non-faulty replicas in x have identical or consistent messages in the wire or *out* variables in y. The next condition requires the same of messages in the wire that are signed by faulty replicas. The final condition says that all requests in the wire in x are also in the wire in y.

To prove that $\mathcal{H}$ is in fact a forward simulation from A'_{gc} to A' one most prove that both of the following are true:

1. For all $x \in \mathit{start}(A'_{gc})$, $\mathcal{H}[x] \cap \mathit{start}(A') \neq \{\}$

2. For all $(x, \pi, x') \in \mathit{trans}(A'_{gc})$, where x is a reachable state of A'_{gc}, and for all $y \in \mathcal{H}[x]$, where y is reachable in A', there exists an execution fragment α of A' starting with y and ending with some $y' \in \mathcal{H}[x']$ such that $\mathit{trace}(\alpha) = \mathit{trace}(\pi)$.

Condition 1 holds because $(x, y) \in \mathcal{H}$ for any initial state x of A'_{gc} and y of A'. It is clear that x and y satisfy the first clause in the definition of $\mathcal{H}$ because the initial value of the variables mentioned in this clause is the same in A'_{gc} and A'. Clauses 2 to 7 are satisfied because $x.in_i$ only contains checkpoint messages, and $y.in_i$, $x.out_i$, $y.out_i$, $x.wire$, and $y.wire$ are empty.

We prove condition 2 by showing it holds for every action of A'_{gc}. We start by defining an auxiliary function $\beta(y, m, a)$ to compute a sequence of actions of A' starting from state y to simulate a receive of message m by an automaton a (where a is either a client or replica identifier):

$\beta(y, m, a) =$
  if $\exists X : ((m, X) \in y.wire)$ then
    if $\exists X : ((m, X) \in y.wire \wedge a \in X)$ then
      $\text{RECEIVE}(m)_a$
    else
      $\text{MISBEHAVE}(m, X, X \cup \{a\}).\ \text{RECEIVE}(m)_a \mid (m, X) \in y.wire$
  else
    if $\exists i : (y.faulty_i = false \wedge m \in y.out_i)$ then
      $\text{SEND}(m, \{a\})_i.\ \text{RECEIVE}(m)_a$
    else
      $\perp$

If $\text{RECEIVE}(m)_a$ is enabled in a state x, there is an m' such that $\beta(y, m', a)$ is defined and the actions in $\beta(y, m', a)$ are enabled for all $y \in \mathcal{H}[x]$, and:

- $m = m'$, if m is not a checkpoint, view-change, or new-view message
- *consistent-vc*(m, m'), if m is a view-change message
- *consistent-nv-set*$(\{m\}, \{m'\})$, if m is a new-view message

This is guaranteed by clauses 5, 6, and 7 in the definition of $\mathcal{H}$.

Now, we proceed by cases proving condition 2 holds for each $\pi \in acts(A'_{gc})$

Non-faulty proxy actions. If π is an action of a non-faulty proxy automaton P_c other than $\text{RECEIVE}(m = \langle \text{REPLY}, v, t, c, i, r\rangle_{\sigma_i})_c$, let α consist of a single π step. For the receive actions, let $\alpha = \beta(y, m, c)$. In either case, when π is enabled in x all the actions in α are also enabled starting from y and an inspection of the code shows that the state relation defined by $\mathcal{H}$ is preserved in all these cases.

Internal channel actions. If π is a $\text{MISBEHAVE}(m, X, X')$ action, there are two cases: if π is not enabled in y, let α be λ; otherwise, let α contain a single π step. In either case, $\mathcal{H}$ is preserved. because π does not add new messages to $x.Wire$.

Receive of request, pre-prepare, prepare, or commit. For actions $\pi = \text{RECEIVE}(m)_i$ where m is a syntactically valid request, pre-prepare, prepare, or commit message, let $\alpha = \beta(y, m, i)$; α transforms y into $y' \in \mathcal{H}[x']$:

- π and α modify *wire* in a way that preserves clauses 5, 6, and 7.
- For receives of request messages, α and π add the same messages to out_i and in_i thereby preserving the state correspondence defined by $\mathcal{H}$.
- For the other message types, the definition of $\mathcal{H}$ and the definition of *in-wv* ensure that when the first if condition is true in x, it is also true in y (because the condition is more restrictive in A'_{gc}, and $x.in_i$ and $y.in_i$ have the same prepare and commit messages with sequence numbers higher than $x.h_i$.) Thus, in this case, the state correspondence defined by $\mathcal{H}$ is preserved. But it is possible for the if condition to be true in y and false in x; this will cause a message to

be added to $y.in_i$ and (possibly) $y.out_i$ that is not added to $x.in_i$ or $x.out_i$. Since this happens only if the sequence number of the message received is lower than or equal to $x.h_i$, the state correspondence is also preserved in this case.

Garbage collection. If $\pi = \text{RECEIVE}(\langle\text{CHECKPOINT}, v, n, d, j\rangle_{\sigma_j})_i$, or $\pi = \text{COLLECT-GARBAGE}_i$, the condition holds when α is λ. It is clear that the condition holds for the first type of action. For the second type, the condition is satisfied because all the messages removed from $x.in_i$ have sequence number lower than or equal to n and the action sets $x.h_i$ to n. The action sets $x.h_i$ to n because it removes all triples with sequence number lower than n from $x.chkpts_i$ and there is a triple with sequence number n in $x.chkpts_i$. The existence of this triple is guaranteed because the precondition for the collect-garbage$_i$ action requires that there is a checkpoint message from i with sequence number n in $x.in_i$ and i only inserts checkpoint messages in in_i when it inserts a corresponding checkpoint in $chkpts_i$.

Receive view-change. If $\pi = \text{RECEIVE}(m = \langle\text{VIEW-CHANGE}, v, n, s, C, P, j\rangle_{\sigma_j})_i$, let $\alpha = \beta(y, m', i)$ such that *consistent-vc*(m, m'). The definition of *consistent-vc* ensures that either both messages are incorrect or both are correct. In the first case, π and α only modify the destination set of the messages in *wire*; otherwise, they both insert the view change message in in_i. In either case, the state correspondence defined by $\mathcal{H}$ is preserved.

Receive new-view. When $\pi = \text{RECEIVE}(m = \langle\text{NEW-VIEW}, v, V, O, N\rangle_{\sigma_j})_i$, we consider two cases. Firstly, if the condition in the outer if is not satisfied, let $\alpha = \beta(y, m', i)$, where *consistent-nv-set*$(\{m\}, \{m'\})$. It is clear that this ensures $y' \in \mathcal{H}[x']$ under the assumption that $y \in \mathcal{H}[x]$. Secondly, if the condition in the outer if is satisfied when π executes in x, let α be the execution of the following sequence of actions of A':

1. The actions in $\beta(y, m' = \langle\text{NEW-VIEW}, v, V', O', N'\rangle_{\sigma_j}, i)$, where *consistent-nv-set*$(\{m\}, \{m'\})$

2. Let C be a sequence of tuples (v_n, R_n, m_n) from $\mathbf{N} \times 2^{\mathcal{R}} \times \mathcal{RM}$ such that the following conditions are true:
   i) $\forall\, n\ :\ (x.\mathit{last\text{-}exec}_i\ <\ n\ \leq\ \mathit{max\text{-}n}(V))$

   ii) $\forall\,(v_n, R_n, m_n)\ :\ (v_n\ <\ v\ \wedge\ |R_n|\ >\ 2f\ \wedge\ \forall\, k\ \in\ R_n\ :\ (\langle\text{COMMIT}, v_n, n, D(m_n), k\rangle_{\sigma_k}\ \in\ x.\mathit{Wire{+}o})$
   $\wedge\ (\exists\, v'\ :\ (\langle\text{PRE-PREPARE}, v', n, m_n\rangle_{\sigma_{primary(v')}}\ \in\ x.\mathit{Wire{+}o})\ \vee\ m_n\ \in\ x.\mathit{Wire{+}o})$

   for each $(v_n, R_n, m_n)\ \in\ C$ in order of increasing n execute:

   a) $\beta(y, c_{n_k} = \langle\text{COMMIT}, v_n, n, D(m_n), k\rangle_{\sigma_k}, i)$, for each $k\ \in\ R_n$

   b) if enabled $\beta(y, p_n = \langle\text{PRE-PREPARE}, v', n, m_n\rangle_{\sigma_{primary(v')}}, i)$ else $\beta(y, m_n, i)$

   c) $\text{EXECUTE}(m_n, v_n, n)_i$

The definition of $\mathcal{H}$ (clauses 1, 4, 5 and 6) ensures that, when the receive of the new-view message executes in y, the condition in the outer if is true exactly when it is satisfied in x. Let y_1 be the state after $\beta(y, m', i)$ executes; we show that when C is empty (i.e., *max-n*$(V) \leq$ *last-exec*$_i$), $y' = y_1 \in \mathcal{H}[x']$. This is true because:

- Both π and $\beta(y, m', i)$ set *view*$_i$ to v, add all the pre-prepares in $O \cup N$ to *in*$_i$, and add consistent new-view messages to *in*$_i$.

- $\beta(y, m', i)$ also adds the pre-prepares in $(O' \cup N') - (O \cup N)$ to *in*$_i$ but this does not violate $\mathcal{H}$ because π ensures that $x'.h_i$ is greater than or equal to the sequence numbers in these pre-prepares.

- Both π and $\beta(y, m', i)$ add prepares to *in*$_i$ and *out*$_i$; $\beta(y, m', i)$ adds all the prepares added by π and some extra prepares whose sequence numbers are less than or equal to $x'.h_i$.

When C is not empty (i.e., *max-n*$(V) >$ *last-exec*$_i$), it is possible that $y_1 \notin \mathcal{H}[x']$ because some of the requests whose execution is reflected in the last checkpoint in x' may not have executed in y_1. The extra actions in α ensure that $y' \in \mathcal{H}[x']$.

We will first show that C is well-defined, i.e., there exists a sequence with one tuple for each n between x.*last-exec*$_i$ and *max-n*(V) that satisfies conditions i) and ii).

Let $m'' = \langle \text{VIEW-CHANGE}, v, \textit{max-n}(V), \langle s, l, t\rangle, C', P, k\rangle_{\sigma_k}$ be the view-change message in V whose checkpoint value, $\langle s, l, t\rangle$, is assigned to $(\textit{val}_i, \textit{last-rep}_i, \textit{last-rep-t}_i)$. Since m'' is correct, C' contains at least $f + 1$ checkpoint messages with sequence number *max-n*(V) and the digest of $\langle s, l, t\rangle$. Therefore, the bound on the number of faulty replicas, and Invariant A.2.3 (condition 2) imply there is a sequence of requests μ_1 such that *committed-Wire*$(s, l, t, \textit{max-n}(V), v, \mu_1)$.

Since by the inductive hypothesis $y \in \mathcal{H}[x]$, all the the commit, pre-prepare and request messages corresponding to μ_1 are also in y.*Wire+o*. Therefore, all the actions in a) and at least one of the actions in b) are enabled starting from y_1 for each n and each $k \in R_n$. Since $v_n < v$ for all the tuples in C, each receive in $\beta(y, c_{n_k}, i)$ will insert c_{n_k} in *in*$_i$. Similarly, the receive of the pre-prepare or request will insert a matching pre-prepare or request in *in*$_i$. This enables execute$(m_n, v_n, n)_i$.

Invariant A.2.3 (condition 1) also asserts that there exists a sequence of requests μ_2 such that *committed-Wire*$(x.\textit{val}_i, x.\textit{last-rep}_i, x.\textit{last-rep-t}_i, x.\textit{last-exec}_i, x.\textit{view}_i, \mu_2)$. Since by the inductive hypothesis $y \in \mathcal{H}[x]$, all the the commit, pre-prepare and request messages corresponding to μ_1 and μ_2 are also in y.*Wire+o*. This and Invariant A.2.4 imply that μ_2 is a prefix of μ_1. Therefore, after the execution of α, *val*$_i$, *last-rep*$_i$, *last-rep-t*$_i$, *last-exec*$_i$ have the same value in x' and y' as required by $\mathcal{H}$.

Send. If $\pi = \text{SEND}(m, X)_i$, let α be:

- A single send$(m, X)_i$ step, if m does not have the CHECKPOINT, VIEW-CHANGE, or NEW-VIEW tag and this action is enabled in y.

- λ, if m has the CHECKPOINT tag or the action is not enabled in y (because the message is already in the channel.)

- A single send$(m', X)_i$ step, if m has the VIEW-CHANGE tag and this action is enabled in y (where *consistent-vc*(m, m').)

- A single send$(m', X)_i$ step, if m has the NEW-VIEW tag and this action is enabled in y (where *consistent-nv-set*$(\{m\}, \{m'\})$.)

Send-pre-prepare and send-commit. If π = SEND-PRE-PREPARE$(m, v, n)_i$ or π = SEND-COMMIT$(m, v, n)_i$, let α contain a single π step. This ensures $y' \in \mathcal{H}[x']$ because these actions are only enabled in x when they are enabled in y, and they insert and remove the same messages from in_i and out_i.

Execute. When π = EXECUTE$(m, v, n)_i$, let α contain a single π step. The action is enabled in y when it is enabled in x because it is only enabled in x for $n > x.h_i$ and $x.in_i$ and $y.in_i$ have the same pre-prepare and commit messages with sequence numbers greater than $x.h_i$ and the same requests. It is easy to see that the state correspondence defined by $\mathcal{H}$ is preserved by inspecting the code.

View-change. If π = VIEW-CHANGE$(v)_i$, let α contain a single π step. The action is enabled in y when it is enabled in x because $view_i$ has the same value in x and y. Both π and α insert view-change messages m and m' (respectively) in in_i and out_i; it is clear that this ensures $y' \in \mathcal{H}[x']$ provided *consistent-vc*(m', m') is true. Clause 2 in the definition of $\mathcal{H}$ ensures that m and m' contain the same messages in the P component for sequence numbers greater than $x.h_i$; therefore, *consistent-vc*(m', m') is true.

Send-new-view. If π = SEND-NEW-VIEW$(v, V)_i$, let α be the execution of the following sequence of actions of A':

1. send-new-view$(v, V')_i$ step, where *consistent-vc-set*(V, V').

2. Let C be a sequence of tuples (v_n, R_n, m_n) from $\mathbf{N} \times 2^{\mathcal{R}} \times \mathcal{RM}$ such that the following conditions are true:
   i) $\forall\, n\ :\ (x.\textit{last-exec}_i\ <\ n\ \leq\ \textit{max-n}(V))$

   ii) $\forall\,(v_n, R_n, m_n)\ :\ (v_n\ <\ v\ \land\ |R_n|\ >\ 2f\ \land\ \forall\, k\ \in\ R_n\ :\ (\langle \text{COMMIT}, v_n, n, D(m_n), k\rangle_{\sigma_k}\ \in\ x.\textit{Wire+o})$
   $\land\ (\exists\, v'\ :\ (\langle \text{PRE-PREPARE}, v', n, m_n\rangle_{\sigma_{\textit{primary}(v')}}\ \in\ x.\textit{Wire+o})\ \lor\ m_n\ \in\ x.\textit{Wire+o})$

   for each $(v_n, R_n, m_n)\ \in\ C$ in order of increasing n execute:

   a) $\beta(y, c_{n_k} = \langle \text{COMMIT}, v_n, n, D(m_n), k\rangle_{\sigma_k}, i)$, for each $k\ \in\ R_n$

   b) if enabled $\beta(y, p_n = \langle \text{PRE-PREPARE}, v', n, m_n\rangle_{\sigma_{\textit{primary}(v')}}, i)$ else $\beta(y, m_n, i)$

   c) EXECUTE$(m_n, v_n, n)_i$

This simulation and the argument why it preserves $\mathcal{H}$ is very similar to the one presented for receives of new-view messages.

Failure. If π = REPLICA-FAILURE$_i$ or π = CLIENT-FAILURE$_i$, let α contain a single π step. It is easy to see that $y' \in \mathcal{H}[x']$.

Actions by faulty nodes. If π is an action of a faulty automaton, let α contain a single π step. The definition of $\mathcal{H}$ ensures that α is enabled in y whenever π is enabled in x. Modifications to the internal state of the faulty automaton cannot violate $\mathcal{H}$. The only actions that could potentially violate $\mathcal{H}$ are sends. But this is not possible because a faulty automaton cannot forge the signature of a non-faulty one. $\square$

Bibliography

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, San Francisco, CA, Oct. 1976.

[Ahm00] S. Ahmed. Private communication, 2000.

[AMP+00] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. Wright. Dynamic Byzantine Quorum Systems. In *International Conference on Dependable Systems and Networks (DSN, FTCS-30 and DCCA-8)*, pages 283–292, New York, New York, June 2000.

[APMR99] L. Alvisi, E. Pierce, D. Malkhi, and M. Reiter. Fault Detection for Byzantine Quorum Systems. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, pages 357–371, San Jose, California, Jan. 1999.

[BEG+94] M. Blum, W. Evans, P. Gemmel, S. Kannan, and M. Naor. Checking the Correctness of Memories. *Algorithmica*, 12:225–244, 1994.

[BHK+99] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO'99*, pages 216–233, 1999.

[BM97] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology – EUROCRYPT' 97*, 1997.

[BR95] M. Bellare and P. Rogaway. Optimal asymmetric encryption - How to encrypt with RSA. In *Advances in Cryptology - EUROCRYPT 94, Lecture Notes in Computer Science, Vol. 950*. Springer-Verlag, 1995.

[BR96] M. Bellare and P. Rogaway. The exact security of digital signatures- How to sign with RSA and Rabin. In *Advances in Cryptology - EUROCRYPT 96, Lecture Notes in Computer Science, Vol. 1070*. Springer-Verlag, 1996.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems*, volume 9(3), Aug. 1991.

[BT85] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–240, 1985.

[CA78] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing, FTCS-8*, pages 3–9, 1978.

[CALM97] M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, Oct. 1997.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *15th International Conference on Fault Tolerant Computing, Ann Arbor, Mi.*, June 1985.

[CHH97] R. Canetti, S. Halevi, and A. Herzberg. Maintaining authenticated communication in the presence of break-ins. In *Proc. of the 1997 ACM Conference on Computers and Communication Security*, 1997.

[CKS00] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, Portland, OR, July 2000.

[CL99a] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[CL99b] M. Castro and B. Liskov. Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography. Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[CL99c] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.

[CL00] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[CR92] R. Canneti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report #92-15, Computer Science Department, Hebrew University, 1992.

[CT00] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. Technical Report 2000-03, University of Arizona, 2000.

[Dam89] I. Damgard. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – Crypto' 89 Proceedings*, number 435 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[DC90] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[DGGS99] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness Failure Detectors: Specification and Implementation. In J. Hlavicka, E. Maehle, and A. Pataricza, editors, *Proceedings of the 3rd European Dependable Computing Conference (EDCC-3)*, pages 71–87. Springer-Verlag, Lecture Notes in Computer Science, Volume 1667, 1999.

[F+97] S. Forrest et al. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[FJL+95] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. H. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6), Aug. 1995.

[FKM00] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[Gei95] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[GGJR99] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure Distributed Storage and Retrieval. *Theoretical Computer Science*, 1999.

[GHM+90] R. Guy, J. Heidemann, W. Mak, J. Page, T., G. Popek, and D. Rothneier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, June 1990.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, Dec. 1979. ACM SIGOPS.

[GK85] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *Database Engineering*, 8(2):63–70, June 1985.

[GM98] J. Garay and Y. Moses. Fully polynomial byzantine agreement for n $\rangle$ 3t processors in t+1 rounds. *SIAM Journal of Computing*, 27(1):247–290, Feb. 1998.

[GMR88] S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen Message Attacks. *SIAM Journal of Computing*, 17(2):281–308, Apr. 1988.

[Gon92] L. Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, Jan. 1992.

[Gra00] J. Gray. FT 101. Talk at the University of California at Berkeley, Nov. 2000.

[HBJ+90] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proceedings of the Workshop on Management of Replicated Data*, Houston, TX, Nov. 1990. IEEE.

[HJJ+97] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proc. of the 1997 ACM Conference on Computers and Communication Security*, 1997.

[HJKY95] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In *Advances in Cryptology – CRYPTO'95*, 1995.

[HKM+88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[HT88] M. Herlihy and J. Tygar. How to make replicated data secure. *Advances in Cryptology (Lecture Notes in Computer Science 293)*, pages 379–391, 1988.

[HW87] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, Jan. 1987.

[Kat97] J. Katcher. PostMark: A New File System Benhmark. Technical Report TR-3022, Network Appliance, Oct. 1997.

[KMMS98] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. of the Hawaii International Conference on System Sciences*, Hawaii, Jan. 1998.

[KP91] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *Theoretical Computer Science*, 4(9):364–373, Nov. 1991.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.

[Lam89] L. Lamport. The Part-Time Parliament. Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 226–238. ACM Press, 1991.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[LZ75] B. Liskov and S. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, SE-1(1), Mar. 1975.

[Mer87] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology - Crypto'87*, number 293 in Lecture Notes in Computer Science, pages 369–378. Springer-Verlag, 1987.

[Min00] R. Minnich. The Linux BIOS Home Page. http://www.acl.lanl.gov/linuxbios, 2000.

[MKKW99] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Kiawah Island, SC, Dec. 1999.

[ML00] B. Murphy and B. Levidow. Windows 2000 dependability. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, New York, NY, June 2000. IEEE.

[MR96a] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. In *Proc. of the 9th Computer Security Foundations Workshop*, pages 9–17, Ireland, June 1996.

[MR96b] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proc. of the 9th Computer Security Foundations Workshop*, pages 9–17, Ireland, June 1996.

[MR97] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of the 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.

[MR98a] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[MR98b] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, Oct. 1998.

[MR00] D. Malkhi and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, Apr. 2000.

[MRL98] D. Malkhi, M. Reiter, and N. Lynch. A Correctness Condition for Memory Shared by Byzantine Processes. Submitted for publication., Sept. 1998.

[MVS00] U. Maheshwari, R. Vingralek, and B. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[OL88] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[Ous90] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proc. of USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[OY91] R. Ostrovsky and M. Yung. How to withstand mobile virus attack. In *Proc. of the 19th Symposium on Principles of Distributed Computing*, pages 51–59. ACM, Oct. 1991.

[Par86] J.-F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. of the 6th International Conference on Distributed Computer Systems*, pages 606–612. IEEE, 1986.

[Pos80] J. Postel. User datagram protocol. DARPA-Internet RFC-768, Aug. 1980.

[Pos81] J. Postel. DoD standard transmition control protocol. DARPA-Internet RFC-793, Sept. 1981.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[Rei94] M. Reiter. Secure Agreement Protocols. In *Proc. of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, Nov. 1994.

[Rei95] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pages 99–110, 1995.

[Rei96] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan. 1996.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Internet RFC-1321, Apr. 1992.

[Rod00] R. Rodrigues. Private communication, 2000.

[S+85] R. Sandberg et al. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[Sat90] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. In *IEEE Computer*, May 1990.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[SDW92] W. Strayer, B. Dempsey, and A. Weaver. *XTP: The Xpress Transfer Protocol*. Addison-Wesley, Reading, Massachusetts, 1992.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SHA94] National Institute of Standards and Technology (NIST). Announcement of Weakness in Secure Hash Standard, 1994.

[Spu00] C. E. Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly and Associates, 2000.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 1983.

[Sul00] B. Sullivan. Inside Europe's cybersleuth central. MSNBC, Oct. 2000.

[TPRZ84] D. B. Terry, M. Painter, D. Riggle, and S. Zhou. The Berkeley Internet Name Domain Server. In *Proceedings USENIX Summer Conference*, Salt Lake City, Utah, June 1984.

[Wie98] M. Wiener. Performance Comparison of Public-Key Cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.

[Yan99] Z. Yang. Byzantine Fault-Tolerant DNS Infrastructure. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.