HQ Replication

by

James Cowling

BCST Adv. Hons. H1M
The University of Sydney (2004)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author . . . .
Department of
Electrical Engineering and Computer Science
May 25, 2007

Certified by . . . .
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

HQ Replication

by

James Cowling

Submitted to the Department of
Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

There are currently two approaches to providing Byzantine-fault-tolerant state machine replication: an agreement-based approach, e.g., BFT, that uses communication between replicas to agree on a proposed ordering of requests, and a quorum-based approach, such as Q/U, in which clients contact replicas directly to optimistically execute operations. Both approaches have shortcomings: the quadratic message cost of inter-replica communication is unnecessary when there is no contention, and Q/U requires a large number of replicas and performs poorly under contention.

This thesis present HQ, a hybrid Byzantine-fault-tolerant state machine replication protocol that overcomes these problems. HQ employs a lightweight quorum-based protocol when there is no contention, but uses BFT to resolve contention when it arises. Furthermore, HQ uses only $3f + 1$ replicas to tolerate f faults, providing optimal resilience to node failures.

We implemented a prototype of HQ, and we compare its performance to BFT and Q/U analytically and experimentally. Additionally, in this work we use a new implementation of BFT designed to scale as the number of faults increases. Our results show that both HQ and our new implementation of BFT scale as f increases; additionally our hybrid approach of using BFT to handle contention works well.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgments

Firstly I'd like to thank my co-authors on the HQ Replication protocol, for their invaluable assistance throughout the project: Daniel Myers, Barbara Liskov, Rodrigo Rodrigues and Liuba Shrira. A great deal of thanks is extended to Barbara, Rodrigo and Liuba, for laying the foundations for HQ before my time at MIT.

The reviewers for our OSDI paper, along with our shepherd Mema Roussopoulos, provided valuable feedback in shaping the current incarnation of the protocol. My co-authors and I would like to thank the developers of Q/U for their cooperation, and the supporters of Emulab, which proved critical in running our experiments.

Thanks go to Dan for a huge effort in coding the BFT module, and much assistance through the first version of the HQ protocol. On a personal level his presence as a friend and lab-mate has been enormously beneficial in my first two years at MIT - as a project partner, sounding board and bicycle technician.

I'd like to thank Jen Carlisle, homeslice and housemate extraordinaire, for the many fun times and moral support.

I'm especially grateful to Teresa, for putting up with a boyfriend living on the other side of the world, and always putting a smile on my face. Also to my parents for their continued support and understanding, especially in the week leading up to this thesis.

Finally I'd like to thank Barbara for her patience, careful thought, insightful comments, and endless discussions on obscure protocol details. This thesis is as much a product of her guidance as it is of my effort.

Contents

Chapter 1

Introduction

Distributed information systems provide shared data access to multiple disparate clients. These systems traditionally consist of a set of servers that store service data, as well as providing functionality for clients to read and modify this shared information. Typical deployments of distributed information systems may be a central company database shared across multiple users, an online medical information system, or a distributed filesystem such as NFS [41].

Multiple redundant servers (replicas) are used in these systems to provide reliability when a subset of servers crash or are disconnected from the network. *State machine replication* protocols [19, 44] are used to ensure that all functioning servers provide a consistent view of the same system state. These protocols provide the illusion that all operations are run at a single central server.

The increasing importance of distributed information services necessitates protocols to ensure correctness and availability of service data, not just when servers crash or are disconnected, but also when they exhibit arbitrary or malicious behavior. These so called *Byzantine faults* [18] may occur due to program bugs or attacks on the system, and introduce additional challenges to reliability. Byzantine fault-tolerant state machine replication protocols, notably the Castro and Liskov BFT protocol [7], address the challenges posed by Byzantine faults.

1.1 Motivation

Initial proposals for Byzantine-fault-tolerant state machine replication [36, 7] rely on a *primary* replica to define a sequential order for client operations, and an *agreement* protocol that runs amongst the replicas to agree on this ordering. Agreement typically takes place in three phases of communication, two of which require all-to-all communication between replicas. Since a minimum of $3f + 1$ replicas are required for the system to tolerate f Byzantine faults [3], an increase in the number of faults tolerated also leads to an increase in the number of replicas. This can pose a scalability problem owing to message load scaling quadratically with the replica set size.

An alternative approach to Byzantine fault tolerance utilizes *quorum* communication, rather than agreement [26]. These protocols shift communication from replica-to-replica to a client-based process. Each client optimistically contacts a quorum of replicas that independently order each operation. These quorums, in the context of Byzantine fault tolerance, are a set of replicas of sufficient size such that any two quorums intersect on at least one non-faulty replica. If the ordering assigned by each replica is consistent, then the operation is able to proceed. If orderings do not match, a repair process is required to reestablish a consistent state.

Quorum-based schemes had previously been limited to a restricted read/blind-write interface [24]. In these systems clients can read system state, or overwrite existing state, but cannot execute operations where the results depend on existing system state. Agreement protocols allow the execution of arbitrary *general operations*, however, that may depend on current system state, rather than just overwriting an object. General operations are far more powerful than blind writes, and allow the implementation of any deterministic service, rather than a simple storage system.

In a recent paper describing the Q/U protocol [1], Abd-El-Malek et al. aim to improve on the scalability of agreement approaches, and show how to adapt Byzantine quorum protocols to implement efficient Byzantine-fault-tolerant state machine replication, providing support for general operations. This is achieved through a client-directed process that requires two rounds of communication between client and replicas when there is no contention and no failures. Q/U is able to perform writes in only one phase when there is no interleaving of writes between clients.

Q/U has two shortcomings, however, that prevent the full benefit of quorum-based systems from being realized. First, it requires a large number of replicas: $5f + 1$ are needed to tolerate f failures, considerably higher than the theoretical minimum of $3f + 1$, as provided in BFT. This increase in the replica set size not only places additional requirements on the number of physical machines and the interconnection fabric, but it also increases the number of possible points of failure in the system; Q/U can only tolerate one-fifth of replica machines being faulty, while BFT can handle one-third.

Second, quorum protocols such as Q/U perform poorly when their optimism fails. The optimistic approach fails when the ordering assigned by replicas to an operation is not consistent. This most often occurs during write *contention*, where multiple clients attempt to perform a write operation at the same time. Q/U clients use exponential back-off to resolve instances of write contention, leading to greatly reduced throughput.

Table 1.1 shows average response time per request in Q/U, for both *fetch* and *increment* operations, the equivalent of general read and write operations respectively in BFT (and also HQ). Readily apparent is the significant performance degradation for contending writes, with more than a 10x increase in delay for contending rather than isolated writes, for four active clients. Performance under write contention is of particular concern, given that such workloads are generated by many applications of interest, such as databases or transactional systems.

1.2 Contributions

This thesis presents the Hybrid Quorum (HQ) replication protocol [9], a new quorum-based protocol for Byzantine fault tolerance that overcomes the limitations of pure agreement-based or quorum-based replication. HQ requires only $3f + 1$ replicas, and combines quorum and agreement-based state machine replication techniques to

		Contending Clients				
		0	**1**	**2**	**3**	**4**
fetch	**Isolated**	320	331	329	326	–
	Contending	–	348	226	339	361
increment	**Isolated**	693	709	700	692	–
	Contending	–	1210	2690	**4930**	**11400**

Table 1.1: Average request response time for Q/U, in μs [1]. *fetch* and *increment* operations are equivalent to reads and writes in HQ respectively.

provide scalable performance as f increases. In the absence of contention, HQ uses a new, lightweight Byzantine quorum protocol in which reads require one round trip of communication between the client and the replicas, and writes require two round trips. When contention occurs, HQ uses the BFT state machine replication algorithm [7] to efficiently order the contending operations.

HQ avoids quadratic message communication costs under non-contention operation, offering low per-message overhead and low latency reads and writes. HQ is able to utilize symmetric key cryptography rather than public key, resulting in reduced CPU overhead. Like Q/U and BFT, HQ supports general operations, and handles Byzantine clients as well as servers. HQ is designed to function in an asynchronous system; Byzantine fault tolerance in synchronous systems [37, 13] is an easier problem, and not a concern of this work.

An additional outcome of this research is a new implementation of BFT. The original implementation of BFT [7] was designed to work well at small f; the new implementation is designed to scale as f grows.

This document presents analytical results for HQ, Q/U, and BFT, and performance experiments for HQ and BFT. These results indicate that both HQ and the new implementation of BFT scale acceptably in the region studied (up to $f = 5$) and that our new approach to resolving contention provides a gradual degradation in throughput as write contention increases.

The contributions of this thesis and the HQ protocol are summarized as follows:

- Development of the first Byzantine Quorum protocol requiring $3f + 1$ replicas, supporting general operations and resilience to Byzantine clients.

- Introduction of a general Hybrid Quorum technique providing contention resolution without performance degradation.
- Optimization of the BFT protocol, realizing increased performance and fault scalability.

1.3 Overview

This thesis describes the design and implementation of a new hybrid approach to Byzantine fault tolerance. Chapter 2 describes the system model and assumptions on failures, communication, and cryptography used by the remainder of this thesis. The basic HQ Replication protocol is introduced in Chapter 3. The mechanisms for contention resolution are discussed in Chapter 4, including the view-change protocol used when the BFT primary fails. Chapter 5 describes a number of optimizations used to improve the performance of the basic algorithm described in earlier chapters, along with extensions to support multi-object transactions. The state transfer protocol is presented in Chapter 6, used to bring slow replicas up to date with the most recent system state. Chapter 7 argues for the safety and liveness of HQ Replication under the assumption of public key cryptography. Chapter 8 then extends HQ Replication to operate with more computationally efficient symmetric key cryptography, and discusses the correctness of the modified protocol. The performance of HQ is examined both analytically and experimentally in Chapter 9, in comparison to both BFT and Q/U. Related work is discussed in Chapter 10, with the thesis concluded in Chapter 11.

Chapter 2

Model and Assumptions

We present here an overview of the system model used in the following chapters. This model defines the operations provided by the system, assumptions on node failures and the communications infrastructure, and the cryptographic primitives available for use by the HQ protocol.

2.1 Operations

HQ and BFT provide support for the execution of *general operations*. These are distinct from simple reads and blind writes to service state, as provided by some previous protocols [26]. Reads and blind writes only allow directly reading or overwriting objects at the server. General operations, however, allow for the execution of complex operations that may depend on current state at the server, and provide a far more powerful interface.

We require all operations to be deterministic, i.e., given a serialized order over a set of operations, each replica should obtain the same result in running each operation, provided they have the same application state.

The terminology *read* and *write* are still used in this thesis to describe the general operations provided by HQ and BFT—*write* operations may modify application state while *reads* may not.

2.2 Failure Model

Our system consists of a set $\mathcal{C} = \{c_1, ..., c_n\}$ of client processes and a set $\mathcal{R} = \{r_1, ..., r_{3f+1}\}$ of $3f + 1$ server processes. Server processes are known as replicas throughout this thesis, as they replicate the server application for reliability.

Clients and servers are classified as either correct or faulty. A correct process is constrained to obey its specification, and follow the HQ or BFT protocol precisely. Faulty processes may deviate arbitrarily from their specification: we assume a Byzantine failure model [17] where nodes may adopt any malicious or arbitrary behavior. We do not differentiate between processes that fail benignly (*fail-stop*) and those suffering from Byzantine faults.

Correct system operation is guaranteed for up to f simultaneously faulty replicas. Transient failures are considered to last until a replica is repaired and has reestablished a copy of the most recent system state. No guarantees are offered beyond f failures, and the system may halt or return incorrect responses to client operations.

No bound is placed on the number of faulty clients. It is assumed that application-level access control is implemented to restrict clients writes to modify only application state for which they have permission. A bad client is able to execute arbitrary write operations on data it has permission to access, but cannot affect other application data nor put the system in an inconsistent state.

2.3 Communication Model

We assume an asynchronous distributed system where nodes are connected by a network infrastructure. We place very weak safety assumptions on this network—it may fail to deliver messages, delay them, duplicate them, corrupt them, deliver them out of order, or forward the contents of messages to other entities. There are no bounds on message delays, or on the time to process and execute operations. We assume that the network is fully connected; given a node identifier, any node can attempt to contact the former directly by sending it a message.

For liveness, we require the use of fair links [47]; if a client keeps retransmitting a request to a correct server, the reply to that request will eventually be received. Liveness for the BFT module used by HQ also requires the liveness conditions assumed by the BFT protocol [7]. Notably, we assume that message delays do not increase exponentially for the lifetime of the system, ensuring that protocol timeouts are eventually higher than message delays. Note that these assumptions are only required for liveness. Safety does not depend on any assumptions about message delivery.

2.4 Cryptography

Strong cryptography is required to ensure correctness in the HQ protocol. Clients and replicas must be able to authenticate their communications to prevent forgeries. We assume that nodes can use unforgeable digital signatures to authenticate messages, using a public key signature scheme such as DSA [31]. We denote a message m signed by node n as $\langle m \rangle_{\sigma_n}$. We assume that no node can send $\langle m \rangle_{\sigma_n}$, either directly or as part of another message, for any value of m, unless it is repeating a previous message or knows n's private key. Any node can verify the integrity of a signature σ_n given the message m and n's public key.

We assume that the public keys for each node are known statically by all clients and replicas, or available through a trusted key distribution authority. Private keys must remain confidential, through the use of a secure cryptographic co-processor or otherwise. If the private key of a node is leaked, the node is considered faulty for the life of the system.

Message Authentication Codes (MACs) [45] are used to establish secure communication between pairs of nodes, despite message transmission on untrusted links. Each pair of nodes shares a secret session key, established via key exchange using public key cryptography. The notation $\langle m \rangle_{\mu_{x,y}}$ is used to describe a message authenticated using the symmetric key shared by nodes x and y.

Chapter 8 shows how to avoid the use of computationally expensive digital signatures under most circumstances. Signatures are replaced with *authenticators*—a vector of MACs between each pair of replicas. Authenticators do not provide the same cryptographic guarantees as signatures, and hence changes must be made to the HQ protocol to accommodate this, discussed in detail in Chapter 8.

We assume the existence of a collision-resistant hash function [39], h. Any node can compute a digest h_m of message m, and it is impossible to find two distinct messages m and m' such that $h_m = h_{m'}$. The hash function is used to avoid sending full copies of data in messages for verification purposes, instead using the digest for verification.

Our cryptographic assumptions are probabilistic, but there exist signature schemes (e.g., RSA [38]) and hash functions (e.g., SHA-1 [33]) for which they are believed to hold with very high probability. Therefore, we will assume they hold with probability 1.0 in remainder of this thesis.

To avoid replay attacks, we tag certain messages with nonces that are signed in replies. We assume that clients use a unique value for each nonce, with no repetition.

Chapter 3

HQ Replication

This chapter describes the operation of the HQ Replication protocol in the absence of write contention. We also introduce the system architecture, along with *grants* and *certificates*, which are the fundamental constructs of the HQ protocol. Further extensions to the base protocol are described in following chapters. We assume the use of public key cryptography in this chapter. We modify the protocol in Chapter 8 to make use of less computationally expensive symmetric key cryptography.

3.1 Overview

HQ Replication introduces a *hybrid* model for state machine replication, providing support for general deterministic operations. Under the normal case of no write contention, it functions as per a quorum agreement scheme [1], albeit with fewer replicas. Under write contention however the protocol adopts an agreement-based approach, based closely on the BFT protocol [7]. The motivation for this design is to exploit the low latency and communication overhead of quorum schemes under normal operation, while employing Byzantine agreement to resolve contending writes without significant performance degradation.

The HQ protocol requires $3f + 1$ replicas to survive f failures, the theoretical minimum. In the absence of failures and contention, write operations execute in two phases, while reads complete in only one phase. Protocol communication is illustrated in Figure 3-1. Each phase consists of the client sending an RPC to each replica, and collecting a quorum of $2f + 1$ responses. Any two quorums intersect at at least $f + 1$ replicas, of which at least 1 is non-faulty. A quorum of matching responses, each authenticated by a different replica, is denoted a *certificate*, and is used to prove agreement in our system.

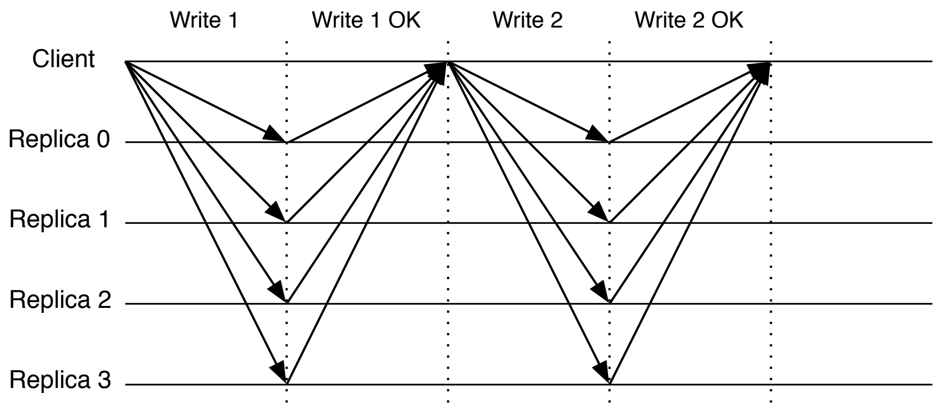

Figure 3-1: HQ Normal-case Communication

The main task of HQ is to ensure that client operations are consistently ordered across a quorum of replicas. The two phases of the quorum protocol facilitate this consistent ordering by assigning a *timestamp* (sequence number) to each operation, and disseminating this ordering to a quorum of replicas. In the first phase a client obtains a timestamp *grant* from each replica, ordering the client write operation with respect to others. In the absence of contention or slow replicas, these grants will match, and the client can compose them into a certificate to convince replicas to execute its operation at the given timestamp. A write is complete once a quorum of replicas has received this certificate and successfully replied to the client.

Timestamps in grants may not match, either when a replica is behind with respect to the latest system state, when there is an operation outstanding that has not yet completed the second phase of the protocol, or when there is write contention. Progress is ensured in the first two cases by a special *writeback* operation, along with *state transfer*, allowing a client to complete a request on behalf of another client, or bring a replica up to date. When there is write contention however, different clients may be assigned the same timestamp from different replicas. Here the set of grants is inconsistent, and we must use BFT to *resolve* the contention. Once contention resolution has completed, each client operation will be consistently ordered across a quorum of replicas, and the system continues under quorum mode.

3.2 System Architecture

The system architecture is illustrated in Figure 3-2. The HQ Replication code resides as a proxy on the client and server machines. Application code on the client executes operation requests via the client proxy, while the server application code is invoked by the server proxy in response to client requests.

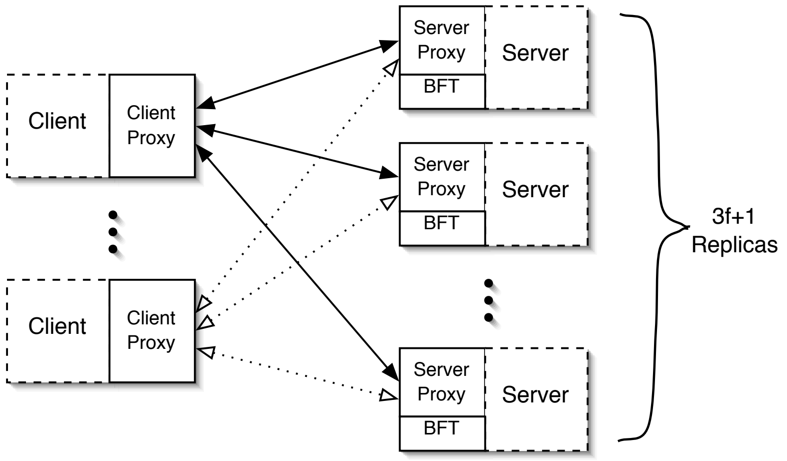

Figure 3-2: System Architecture

The replicated system appears to the client application code as a single fault-tolerant server. The client proxy ensures that all application requests are retried until successful, and that the application only sees the results of operations that have committed. Likewise, the server application code considers the system as a simple client-server architecture, and is insulated from the agreement process by the server proxy. The server application code must support three specific properties, however:

- The server application must support one level of undo, to reverse the effects of the most recent operation. This is required to support reordering during contention resolution, as discussed in Chapter 4.

- All operations executed by the application must be deterministic, to ensure consistent state across a quorum of replicas.
- Application state must be available to the server proxy to facilitate state transfer in Chapter 6. The proxy only needs access to this state, not knowledge of its semantics.

The server proxies make use of the BFT state machine replication protocol [7] to resolve write contention. BFT is not involved in the absence of contention.

3.3 Normal-case Processing

HQ proceeds without the use of BFT under the normal case of no write contention and no failures. We present an unoptimized version of the non-contention protocol in this section, with optimizations discussed in Section 5.4.

HQ supports multiple server objects, each serviced by its own instance of the quorum protocol. Multiple objects are useful in providing finer write granularity, to avoid contention through false sharing. We provide a scheme for running transactions across multiple objects in Section 5.6, but assume in this section that each operation concerns only a single object. Clients may only have one outstanding write operation on a particular object at a particular time[1]. Clients number each write request sequentially, to avoid the replicas running the same operation more than once. Replicas maintain state storing the latest sequence number for each client; this sequence number may be requested by clients if necessary after a reboot.

3.3.1 Grants and Certificates

Grants represent a promise from a replica to execute an operation at a given timestamp, provided there is agreement on that ordering by a quorum of replicas. A grant

[1]As we support general operations, clients may batch multiple operations into a single logical write operation, to avoid running the protocol multiple times.

takes the form $\langle cid, oid, op\#, h, vs, ts, rid\rangle_{\sigma_r}$, where each grant is signed by a replica with with id *rid*. It states that the replica has granted the client with id *cid* the right to run an operation numbered *op#*, on object *oid* at timestamp *ts*. *h* is a hash covering the operation, *cid*, *oid* and *op#*; it is used to check whether two grants refer to the same client operation. *vs* is the *viewstamp* used by BFT when performing contention resolution, as described in Chapter 4. A grant is *later*, or more recent, than another if it has a higher viewstamp, or timestamp for grants in the same view.

A *write certificate* is simply a quorum of grants. It is valid if the grants are from different replicas, are correctly authenticated (signed), and all other fields in the grants are identical. A valid certificate proves that a quorum of replicas agree to an ordering of a specific operation, and precludes the existence of any conflicting certificate for the same viewstamp and timestamp. A replica will execute an operation at a certain timestamp given a certificate with a current or more recent viewstamp, regardless of whether it sent a matching grant itself. We use the notation *c.cid*, *c.ts*, etc., to denote the corresponding components of write certificate *c*.

We say that two write certificates *match* if their constituent grants are identical in all fiends except *rid*. Two certificates may hence contain grants from different quorums of replicas, yet be functionally identical.

3.3.2 Client protocol

Write Protocol

Write requests are performed in two phases. In the first phase the client obtains a grant from a quorum of replicas, each authorizing the client to perform the operation at a specific timestamp. In the second phase the client combines a quorum of matching grants into a certificate and sends these to the replicas, providing proof of a consistent ordering. We provide mechanisms for progress when this ordering is not consistent across replicas.

Pseudocode for the basic client write protocol is given in Figure 3-3, with the protocol discussed in detail in the following sections.

Protocol at client c to execute write operation *op* with arguments *args*.

- Phase 1.

  1. Send $\langle$WRITE-1, *cid, oid, op#, op*$\rangle_{\sigma_c}$ to all replicas
  2. Wait for a quorum of valid (well-formed and correctly authenticated) replies of the form $\langle$WRITE-1-OK, *grantTS, currentC*$\rangle$ or $\langle$WRITE-1-REFUSED, *grantTS, cid, oid, op#, currentC*$\rangle_{\mu_{c,r}}$, or a single valid $\langle$WRITE-2-ANS, *result, currentC, rid*$\rangle_{\mu_{c,r}}$ message.
  3. Proceed according to one of the five conditions:
     (a) Each $reply_i$ in quorum is a WRITE-1-OK, all $grantTS_i.ts$'s are equal and all $grantTS_i.vs$'s are equal:
     Proceed to phase 2 with certificate built from $grantTS_i$'s.
     (b) Each $reply_i$ in quorum is a WRITE-1-REFUSED and all $grantTS_i.ts$'s, all $grantTS_i.vs$'s, all $grantTS_i.h$'s, and all $grantTS_i.cid$'s are equal:
     Perform a writeback on behalf of replica $grantTS_i.cid$.
     (c) $grantTS_i.ts$'s or $grantTS_i.vs$'s not all equal:
     Perform a writeback to slow replicas with certificate with highest *vs* and *ts*.
     (d) A WRITE-2-ANS message is received:
     Proceed to phase 2 with certificate in WRITE-2-ANS message.
     (e) All $grantTS_i.ts$'s and $grantTS_i.ts$'s equal, but $grantTS_i.cid$'s or $grantTS_i.h$'s are not all equal:
     Contention encountered, resolve using BFT.

- Phase 2.

  1. Send $\langle$WRITE-2, *writeC*$\rangle$ to all replicas, with *writeC* certificate from Phase 1.
  2. Wait for a quorum of valid $\langle$WRITE-2-ANS, *result, currentC, rid*$\rangle_{\mu_{c,r}}$ with matching *results* and *currentC*'s.
  3. Return *result* to the client application.

Figure 3-3: Basic client write protocol.

Write Phase 1 A client sends a write phase-1 request to all replicas in the form $\langle$WRITE-1, *cid, oid, op#, op*$\rangle_{\sigma_c}$. We note again that *op#* specifies the operation number for this operation, incremented by the client each time it performs a write, and used by the replicas to avoid replays. A separate invocation of the HQ protocol exists for each object the operation is executed upon, as defined by *oid*.

Replicas may reply with a number of different responses, depending on their current state:

- $\langle$WRITE-1-OK, *grantTS, currentC*$\rangle$, if the replica has no outstanding grants and has granted the next timestamp to this client. *grantTS* is a grant for the client's operation, and *currentC* is a certificate for the most recent committed write at the replica.

- $\langle$WRITE-1-REFUSED, *grantTS, cid, oid, op#, currentC*$\rangle_{\mu_{c,r}}$, if the replica already has an outstanding grant for a different client. The reply contains the grant for the other client, along with the information in the current client's request, to prevent replays.
- $\langle$WRITE-2-ANS, *result, currentC, rid*$\rangle_{\mu_{c,r}}$, if the client's write has already been executed. This can happen if the client was slow and a different client completed the write on its behalf, using the writeback operation discussed below. Here *currentC* is the certificate proving the client's write committed at a given timestamp.

Invalid responses are discarded—those where $grantTS.ts$ is not equal to $currentC.ts+1$, $grantTS.vs < currentC.vs$, *currentC* is not a legitimate certificate, $\mu_{c,r}$ is invalid, or the grants do not match the original client request. After waiting for $2f+1$ valid responses from different replicas, the client processes them as follows:

Success If the client receives a quorum $(2f+1)$ of WRITE-1-OK responses for the same viewstamp and timestamp, then there is a consistent ordering for the operation. It combines each *grantTS* into a certificate and uses this certificate to execute phase 2 of the write protocol, discussed in the following section.

Current Write Outstanding If the client receives a quorum of WRITE-1-REFUSED responses for the same viewstamp, timestamp, and hash, but for a different single *cid*, some other client has a currently outstanding write. This other client should perform a phase 2 write to complete the operation, but to facilitate progress in the presence of failed or slow clients, we allow this client to complete the second phase on its behalf. We term this procedure a *writeback*, where a client advances the state of the replicas on behalf of another client, while piggybacking its own request to be executed once the writeback is complete. The client sends a $\langle$WRITEBACKWRITE, *writeC, w1*⟩ message to the replicas, where *writeC* is a certificate formed from the grants for the other client, and $w1$ is a copy of its original write request. The replicas will reply with their responses to the WRITE-1, and the client repeats the phase 1 response processing.

Slow Replicas If the client receives grants with differing viewstamps or timestamps, it means that the replicas aren't all at the same state, and some are behind in executing operations. The client uses a writeback operation to bring the slow replicas up to date with the latest system state, in doing so soliciting new grants from these slow replicas. It again sends a ⟨WRITEBACKWRITE, *writeC, w1*⟩, where *writeC* is the certificate sent in the response with the highest viewstamp and timestamp. This message is sent only to replicas that are behind with respect to this latest state.

Write Completed If the client receives any WRITE-2-ANS response then the phase 2 write has already been executed on its behalf, courtesy of a writeback operation by a different client. It uses the certificate in this response to execute phase 2 at all other replicas, to solicit their phase 2 responses.

Contention If the client receives a quorum of responses containing grants with the same viewstamp and timestamp, but otherwise different (notably for different clients or operations), then contention has occurred. Here the grants are inconsistent, and must be resolved by an instance of BFT. The client forms these inconsistent grants into a *conflict certificate* and sends these to the replicas in a RESOLVE request, as discussed in Chapter 4. The responses to a resolve request are identical to those for a WRITE-1, and are handled by the protocol above.

Write Phase 2 Following the completion of phase 1 of the write protocol, the client holds a quorum of matching grants to execute its operation at a given timestamp and viewstamp. It combines these into a *writeC* certificate, and sends a ⟨WRITE-2, *writeC*⟩ request to all replicas. The client waits for a quorum of valid responses of the form ⟨WRITE-2-ANS, *result, currentC, rid*$\rangle_{\mu_{c,r}}$, with matching *result*s and certificates.

The result is then returned to the calling application and the write is complete. The client will receive this quorum of matching WRITE-2-ANS responses unless there is contention, with this case discussed in Chapter 4.

Read Protocol

The client read protocol completes in a single phase in the absence of concurrent writes. It consists of simply sending a request for data to all replicas, and returning to the application if a quorum of matching responses are received.

The client sends a $\langle \text{READ}, \textit{cid, oid, op, nonce}\rangle_{\mu_{c,r}}$ request to the replicas. The *nonce* allows a client to distinguish responses between requests, and prevent replay attacks. The $\mu_{c,r}$ MAC authenticates client c to each replica r, and is included to support read access control at the replicas if desired[2]. The response to a read has form $\langle \text{READ-ANS}, \textit{result, nonce, currentC, rid}\rangle_{\mu_{c,r}}$, where *nonce* matches that in the client request, and *currentC* is the certificate supporting the latest known state at the replica. The MAC in the response ensures that a malicious replica does not spoof responses on behalf of others.

Once a quorum of valid matching replies has been received, the client can return the result to the calling application. If the responses contain differing timestamps or viewstamps, the client needs to bring slow replicas up to date to receive a matching quorum. This is done with a writeback operation, where the client sends a $\langle \text{WRITEBACKREAD}, \textit{writeC, cid, oid, op, nonce}\rangle_{\mu_{c,r}}$ message to the slow replicas. *writeC* is the highest certificate received in the read responses. Replicas will advance their state to that reflected in *writeC*, and respond again to the read request.

A client may have multiple reads outstanding, and execute reads concurrently with a pending write operation. A read operation may also be executed as a write, to ensure progress where there are continual write operations that hinder the retrieval of a matching quorum of read responses.

[2]We note that a bad replica may always leak user data, and hence access control cannot be relied upon for privacy.

Protocol at replica r to handle write protocol messages.

- Phase 1. On receiving $\langle \textsc{write-1}, \textit{cid, oid, op\#, op} \rangle_{\sigma_c}$:
    1. If request is invalid (incorrectly authenticated or $\textit{op\#} < \textit{oldOps}[\textit{cid}].\textit{op\#}$), discard request. Resend $\langle \textsc{write-2-ans}, \textit{oldOps}[\textit{cid}].\textit{result}, \textit{oldOps}[\textit{cid}].\textit{currentC}, \textit{rid} \rangle_{\mu_{c,r}}$ if $\textit{op\#} = \textit{oldOps}[\textit{cid}].\textit{op\#}$.
    2. If $\textit{grantTS} = \textit{null}$, set $\textit{grantTS} = \langle \textit{cid, oid, op\#, h, vs, currentC.ts+1, rid} \rangle_{\sigma_r}$, and reply with $\langle \textsc{write-1-ok}, \textit{grantTS, currentC} \rangle$.
    3. If $\textit{grantTS} \neq \textit{null}$, reply with $\langle \textsc{write-1-refused}, \textit{grantTS, cid, oid, op\#, currentC} \rangle_{\mu_{c,r}}$.
- Phase 2. On receiving $\langle \textsc{write-2}, \textit{writeC} \rangle$
    1. If request is invalid (incorrectly authenticated or $\textit{writeC.op\#} < \textit{oldOps}[\textit{writeC.cid}].\textit{op\#}$) discard request. Resend $\langle \textsc{write-2-ans}, \textit{oldOps}[\textit{writeC.cid}].\textit{result}, \textit{oldOps}[\textit{writeC.cid}].\textit{currentC}, \textit{rid} \rangle_{\mu_{c,r}}$ if $\textit{writeC.op\#} = \textit{oldOps}[\textit{writeC.cid}].\textit{op\#}$.
    2. If $\textit{writeC.ts} > \textit{currentC.ts} + 1$, $\textit{writeC.vs} > \textit{currentC.vs}$, or no operation corresponding to *writeC.h* exists in *ops*, perform state transfer and proceed to step 3.
    3. Execute request corresponding to *writeC.h* from *ops*, to get *result*.
    4. Store *op#*, *writeC* and *result* in *oldOps*[*cid*]. Set *currentC* to *writeC*. Set *grantTS* to *null*. Clear *ops* except for WRITE-1 corresponding to just-executed operation.
    5. Respond to client with $\langle \textsc{write-2-ok}, \textit{result, currentC, rid} \rangle_{\mu_{c,r}}$

Figure 3-4: Basic replica write protocol.

3.3.3 Replica protocol

Here we detail the processing at each replica in executing the HQ protocol in the absence of contention. Replicas discard any invalid requests, i.e., those that contain invalid certificates or are improperly signed, and process remaining requests as described in this section. Pseudocode for the basic replica write protocol is given in Figure 3-4.

Replica State

Replicas must maintain state per object to ensure the correct serialization of client requests. The protocol state required in the absence of contention is as follows:

currentC A certificate supporting the latest state of the object. This is the certificate for the most recently committed write operation.

grantTS A grant for the next timestamp. If a client has a currently outstanding write, this will be the grant sent to that client in the WRITE-1-OK reply, otherwise it will be null.

vs The current viewstamp. It is advanced each time the system performs contention resolution or a view change occurs.

ops The set of WRITE-1 requests that are currently active. This includes the request that was granted the current timestamp, and any request that has been refused, along with the most recently executed request.

oldOps A table storing the results of the most recent write for each authorized client. This table maps *cid* to *op#*, along with the *result* and *currentC* sent in the WRITE-2-ANS response.

Write Phase 1

As discussed previously, write requests take the form $\langle$WRITE-1, *cid, oid, op#, op*$\rangle_{\sigma_c}$. If *op#* = *oldOps*[*cid*].*op#*, the request is a duplicate of the client's most recently committed write; the previous WRITE-2-ANS reply is retrieved from the client's entry in *oldOps*, and resent to the client. If *op#* < *oldOps*[*cid*].*op#*, the request is old, and is discarded by the replica. If the request is contained in *ops* then it is a duplicate of the currently outstanding write, and the replica responds with its previous WRITE-1-OK or WRITE-1-REFUSED response. This response can be recomputed based on the standard phase 1 processing, as discussed next.

The replica appends any new valid request to *ops*. If $grantTS = null$, there is no currently outstanding write, and the replica can grant the next timestamp to the client. It sets $grantTS = \langle cid, oid, op\#, h, vs, currentC.ts{+}1, rid\rangle_{\sigma_r}$. The grant and most recent certificate are sent to the client in a $\langle$WRITE-1-OK, *grantTS, currentC*$\rangle$ message.

If $grantTS \neq null$ then there is a currently outstanding write. The replica sends a $\langle$WRITE-1-REFUSED, *grantTS, cid, oid, op#, currentC*$\rangle_{\mu_{c,r}}$ message to the client, where *grantTS* is the grant that was assigned to the client with the outstanding write, *currentC* is the most recent write certificate, and the remaining fields reflect the current client request.

Write Phase 2

Phase 2 write requests consist only of a single write certificate, sent as $\langle$WRITE-2, *writeC*$\rangle$. Unlike WRITE-1, a WRITE-2 may come from any node, not specifically the client whose request is ordered by the certificate. This is used to ensure progress with slow or failed clients, as previously mentioned, and is possible because the certificate itself identifies the *cid* of the client that requested the operation, along with the *op#*, *oid* and hash h uniquely identifying the operation. Note also that a replica will process a WRITE-2 operation regardless of whether it granted to the client in the certificate—the certificate is enough to prove a quorum of replicas agree on the ordering, regardless of the grants of any specific replica.

As with phase 1, old and duplicate requests are detected based on the *op#* in *writeC*, by comparison with *oldOps*[*cid*].*op#*. Old requests are discarded while duplicate requests for the most recently committed write are serviced by returning the WRITE-2-ANS response in *oldOps*[*cid*].

If the request is new and the certificate valid, the replica will make an up-call to the server application code to execute the operation, as stored in *ops*. The replica state must be current with respect to the requested operation—if $writeC.ts > currentC.ts + 1$ or $writeC.vs > vs$ then the replica is behind and must perform state transfer (described in Section 6) before executing the operation, moving to the most recent view.

After receiving the results of the operation from the server application, the replica updates its protocol state information. *oldOps*[*cid*] is updated to reflect the *op#* and *writeC* in the request, along with the operation result. *currentC* is set to *writeC* and *grantTS* to *null* to reflect the latest system state, with *ops* cleared to contain only the request just executed. The inclusion of the just-executed request in *ops* ensures that a committed operation will persist through a concurrent instance of contention resolution, as described in Section 4.4.2. The replica then replies to the client with $\langle$WRITE-2-ANS, *result, currentC, rid*$\rangle_{\mu_{c,r}}$.

Read Protocol

No specific client state is required to execute read operations, apart from correctly authenticating the client. Upon receiving a $\langle$READ, *cid, oid, op, nonce*$\rangle_{\mu_{c,r}}$ request, the replica performs an up-call to the server application code, to execute the read operation *op*. The return value from this call is sent to the client in a $\langle$READ-ANS, *result, nonce, currentC, rid*$\rangle_{\mu_{c,r}}$ message, where *nonce* echos that in the client request.

Writeback

A $\langle$WRITEBACKREAD, *writeC, cid, oid, op, nonce*$\rangle_{\mu_{c,r}}$ request or $\langle$WRITEBACKWRITE, *writeC, w1*$\rangle$ are simply a write operation paired with a subsequent read or write request. The replica first processes the *writeC* certificate as it would a WRITE-2 request, transferring state if necessary, but suppresses any response to the client indicated in the certificate. This response will be returned to the originating client if it attempts to complete its operation at a later time.

Following the writeback, the replica processes the bundled READ or WRITE operation as normal, returning the result to the client.

Chapter 4

Contention Resolution

One of the major strengths of the HQ Replication protocol, with respect to existing optimistic quorum replication protocols [1], is its ability to handle contending writes without a significant degradation in throughput. HQ achieves this through the use of the BFT protocol, which is used as a subroutine to resolve instances of write contention. This chapter describes the use of the BFT module within HQ.

As in the previous chapter, we assume the use of signatures and public key cryptography. Chapter 8 extends the contention resolution protocol to use faster symmetric key cryptography. We also assume that the primary BFT replica is non-faulty, Section 4.7 presents the view change protocol employed when this primary fails.

4.1 BFT Overview

We begin this chapter with a brief overview of the BFT protocol. The high-level operation of BFT is described here to provide background for the contention resolution protocol; further details on BFT are available in [7].

BFT is a Byzantine fault tolerant state machine replication protocol [15, 44]. It is used to reach agreement on a series of client operations, amongst a set of at least $2f+1$ replicas. Agreement is coordinated by a replica that is denoted the *primary*. The term *view* is used to describe a system configuration with a particular primary. The BFT protocol moves through a series of views [34], each denoted by a *view number*;

the primary for a given view determined based on the view number. Replicas remain in the current view unless the primary replica is suspected of being faulty. If the primary is not coordinating the agreement protocol in a correct and timely way, the other replicas will execute a *view change*, incrementing the view number and moving to a new view.

Clients send write requests to all replicas, and wait for $f + 1$ matching responses. The replicas must reach agreement on a sequence number for each operation in order, before executing the write operation and responding to the client. Agreement proceeds in three phases:

pre-prepare The primary assigns a sequence number to the operation, and broadcasts this sequence number, along with the current view number and operation hash, to all replicas in a PRE-PREPARE message.

prepare If a non-primary (backup) replica receives a new valid pre-prepare, it broadcasts a PREPARE message to all replicas, indicating that it accepts the assigned sequence number.

commit If a replica receives $2f+1$ matching prepare messages, then at least $f+1$ honest replicas agree on the sequence number. It broadcasts a COMMIT message to all replicas for the operation and sequence number.

A replica waits for $2f + 1$ matching COMMIT messages, at which point it may execute the operation, and send the result to the client. The three phases are required (unlike two-phase agreement in non-Byzantine agreement protocols [34, 16]), to ensure that knowledge of committed operations persists when view changes occur during agreement.

Replicas maintain a log of agreement messages to facilitate state transfer to slow replicas (replicas that did not participate in the most recent round of agreement), and to ensure transfer of potentially committed operations to new views. A CHECKPOINT protocol is used to truncate this log. *Checkpoints* are snapshots of the application state, taken at a particular sequence number, and signed by $f + 1$ replicas. Replicas exchange signed digests of these checkpoints at regular intervals. A checkpoint signed by $f+1$ replicas may be used to prove the validity of an application state snapshot, and allows log entries for previous sequence numbers to be truncated. We discuss the state management protocol in more detail in Section 6.3.

4.2 Contention

Write contention occurs when multiple clients attempt concurrent writes on the same object, and are assigned the same timestamp. It is noted by clients when receiving a quorum of conflicting grants in response to a WRITE-1 request, as described in Section 3.3.2.

Not all instances of concurrent writes result in contention under the HQ protocol—if a single client is assigned a quorum of matching grants, then competing clients will use a writeback request to complete their writes, without the need for contention resolution. Moreover, contention may be observed where there are in fact no clients competing for the same timestamp. This can occur if a faulty client sends different requests to each replica, causing a mismatch in the resultant grants, or a bad replica responds with a new grant for an old or non-existent client request. In both cases safety is not compromised, but a malicious entity can force the protocol to take the slower agreement-based path of contention resolution.

Contention resolution utilizes BFT to reach agreement on the latest system state, and set of conflicting writes. Armed with this information, the replicas deterministically order and execute the conflicting operations. The resolution process guarantees:

1. if the write second phase has completed for an operation o at timestamp t, then o will continue to be assigned to t.

2. if a client obtains a certificate to run o at t, but has not yet completed the second phase of the write protocol, o will run at some timestamp $\geq t$.

It is possible in the second case that some replicas have executed o at timestamp t, but the operation later commits at a higher timestamp. Hence replicas need to provide one level of undo, maintaining a backup to reverse the effects of the most recently executed operation if it is reordered during contention resolution. While the server application code must support this undo functionality, the undo is never visible to the application code on the client side. The client application never sees the results of an operation until the write protocol is complete, when a quorum of matching WRITE-2-ANS responses has been received, at which point the operation is committed and will retain its order.

4.3 Resolve Requests

Clients request contention resolution by sending a ⟨RESOLVE, *conflictC, w1*⟩ message to all replicas. Here $conflictC$ is a *conflict certificate* formed from the grants it received in response to its WRITE-1 request. A conflict certificate is analogous to a write certificate, except that it proves that a quorum of replicas *disagree* on the client or operation for a particular timestamp and viewstamp, rather than the agreement proved by a write certificate. $w1$ is the client's original WRITE-1 request that led to the conflict, included to ensure the client's operation is executed as part of the contention resolution process.

4.4 Replica Protocol

Contention resolution makes use of the BFT state-machine replication protocol [7], which runs alongside HQ on the replicas, as in Figure 3-2. One of the replicas is denoted the BFT *primary*, which is tracked by the HQ server code by the same mechanisms as a client in BFT. The clients of the BFT protocol in our system are not the HQ clients, but the HQ replicas themselves. Rather than using BFT to order client operations, as was its original function, we use it to reach agreement on a consistent set of replica state information. With consistent knowledge of all conflicting operations, each replica can deterministically order the operations—thus a single round of BFT orders and commits all currently contending writes.

The protocol for using BFT to resolve contention is discussed in detail in the following sections.

4.4.1 Additional State

We need to add additional state to replicas to support contention resolution, in addition to the local state maintained by the BFT protocol itself [7]. For each object in the system, replicas maintain:

conflictC The conflict certificate that started the current instance of contention resolution. If contention resolution is not active this will be null.

backupC The write certificate for the previous write operation. This is the certificate for the write before the operation represented by *currentC*.

prevOp The information previously stored in *oldOps* for the client whose write request was most recently executed.

The latter two state items provide backups of the state, before the most recently executed operation. They are used to restore the previous state if the latest operation is undone during contention resolution.

We delayed discussion of viewstamps, first mentioned in Chapter 3, until now. Viewstamps are used to provide compatibility with the BFT protocol, and represent the ordering of operations within BFT. A viewstamp is a pair consisting of the current BFT view number, along with the sequence number assigned by BFT to the most recently executed operation. View number takes precedence over BFT sequence number in ordering, i.e., the viewstamp with the highest view number is considered most recent, with BFT sequence number used to define ordering where the view numbers are equal. Viewstamps are included in grants to ensure that each round of the HQ quorum protocol runs in the same BFT view.

4.4.2 Request Processing

A replica may execute only one concurrent RESOLVE at a time. Resolve status is indicated by $conflictC$; if it is non-null then contention resolution is in process and we deem the replica *frozen*. A frozen replica will not respond to any further RESOLVE requests or client write operations; instead these will be buffered and processed after resolution is complete. Write requests are postponed to ensure that replica state does not change during resolution. READ requests may be processed since they do not impact protocol state.

A non-frozen replica processes a $\langle$RESOLVE, *clientConflictC, w1*$\rangle$ request as follows:

1. If $clientConflictC.ts > currentC.ts + 1$ or $clientConflictC.vs > currentC.vs$, then the replica is behind with respect to the viewstamp and timestamp where contention has occurred. It first brings itself up to date using state transfer (described in Chapter 6), updating *currentC*, and then continues processing the RESOLVE request from step 2.

2. If *currentC* is more recent than $clientConflictC$, the conflict has already been resolved. This is also the case if the viewstamps and timestamps in the two certificates match, but the client request has already been executed according to *oldOps*. The replica ceases processing the RESOLVE and instead handles $w1$ the same way as a WRITE-1 request, as described in Section 3.3.3.

3. If *currentC* is not more recent than $clientConflictC$, the replica stores $clientConflictC$ in $conflictC$, entering *frozen* mode. It adds $w1$ to *ops* if it is not already there, to ensure the client write is ordered as part of resolution. $w1$ might not already exist in *ops* if the client communicated its original write request with other replicas than this one. The replica builds a signed $\langle$START, *conflictC, ops, currentC, grantTS*$\rangle_{\sigma_r}$ message and sends it to the replica that is the current primary in the BFT protocol. This message is a summary of the current protocol state known at the replica, it ensures that all replicas in the

resolution process will be aware of all conflicting operations, the highest current system state, and any outstanding grants across the $2f + 1$ participating replicas.

Contention resolution may fail to proceed after a replica sends a START message to the primary, either when the primary is faulty, or if the client only sent the RESOLVE request to a subset of replicas. The replica detects this by a simple timeout mechanism, and must rectify the situation to ensure it does not remain frozen indefinitely. The replica broadcasts the RESOLVE request to all replicas in the system. If they have not previously acted upon the resolve request they will enter frozen state and send a START message to the primary. This same mechanism is used to trigger a view change where the primary is faulty, detailed in Section 4.7.

Note that if a replica receives a RESOLVE message from another replica, it must also set a timer and broadcast the request if the timer expires. This addresses the scenario where a bad replica selectively freezes other replicas by forwarding them a RESOLVE request in isolation.

4.4.3 BFT Operation

The HQ proxy running on the primary waits to receive a quorum of valid START messages, properly signed and containing legitimate certificates, including one from itself. It then combines the quorum of START messages into a structure called *startQ*. The HQ code at the primary acts as a client of the BFT module running on the same machine, and sends *startQ* as an operation to run on the BFT service. This *startQ* is used as the argument to BFT, and will provide global knowledge of the state to all replicas that participate in BFT.

BFT will order the *startQ* operation with respect to any earlier request to resolve contention, and conducts agreement among the replicas on this operation. Following agreement, each replica will pass *startQ* to the HQ proxy code running on the corresponding machine. This takes the place of executing an operation in the standard BFT protocol—unlike standard BFT, no execution response is send to the BFT client. The BFT module also passes the current viewstamp to the HQ proxy, which has been incremented due to the running of BFT.

4.4.4 State Processing

Each replica participating in BFT is provided with a consistent view of system state and all conflicting operations via *startQ*. This state must then be processed to order the conflicting operations, and to resolve the contention.

The processing described in this section involves an additional phase of communication, to obtain grants for each operation. We are able to avoid this communication by slightly modifying the BFT protocol; this is presented as an optimization in Section 5.4.

In the following discussion we use the notation *startQ.currentC*, *startQ.ops*, etc., to denote the corresponding list of components in the START messages in *startQ*.

1. If *startQ* doesn't contain a quorum of correctly signed START messages then the primary must be faulty. The replica aborts processing of the new state, and requests a view change, discussed in Section 4.7. When the view change is complete, the replica sends its START message to the new primary, and the BFT process is repeated.

2. The replica determines the most recent operation state, denoted as C, and brings itself up to date with respect to this certificate:

    (a) The replica first checks if a single operation already accumulated a quorum of grants for the current timestamp. A client may be assigned a quorum of matching grants but still detect contention since it can only wait for $2f+1$ write phase 1 responses, some of which may conflict. This is exhibited in the shared state by a quorum of matching grants in *startQ.grantTS*. In this case, C is set to the certificate formed from the matching grants.

    (b) If *startQ.grantTS* does not form a certificate, the latest valid certificate is chosen from *startQ.currentC*, and stored as C.

(c) The replica determines if its current state *currentC* is more recent than C. This can occur if the replica did not contribute to the quorum of START messages in *startQ*, and those that did had not yet received the WRITE-2 request for the operation the replica most recently executed. It must then undo the most recent operation; this is safe because the operation could not have committed. It does so by making a call to the server application code to undo the most recent operation on the application data, and then restores its previous protocol state by setting *currentC* to *backupC*, and replacing the corresponding client data in *oldOps* with that in *prevOp*.

(d) Finally the replica brings itself up to date by executing the operation identified by C, if it hasn't already done so. The replica must execute all operations in order, so may need to run earlier operations if it is behind—it obtains the necessary data via state transfer (described in Chapter 6) from other replicas. Operations are executed as per normal operation, with *oldOps* and *currentC* updated to reflect each new operation, although no replies are sent to clients. Following this phase, all replicas participating in contention resolution will have identical *oldOps* and *currentC*.

3. The replica builds an ordered list L of all conflicting operations that need to be executed. L contains all non-duplicate operations in *startQ.ops*, inserted in some deterministic order, such as ordered on the *cid* of the client that requested each operation. Duplicate operations are detected based on entries in *oldOps*, and ignored. Multiple operations from the same (faulty) client are also ignored, with one operation per client selected based on a deterministic means, such as the lowest operation hash. Entries in *startQ.ops* are signed by clients and hence cannot be forged.

4. The replica needs a certificate to support the execution of each operation in L. This is obtained using an additional phase of communication as follows. (Section 5.4 provides an optimization to piggyback this phase along with BFT operation).

- A grant is created for each operation in L, with viewstamp vs set to that returned by BFT, and timestamp corresponding to the order of each operation in L. This set of grants is sent to all replicas.
- The replica waits for $2f + 1$ matching grants for each operation in L from other replicas, including one from itself. These are combined into a certificate for each operation.

5. The operations in L are executed in order, updating *ops*, *oldOps*, $grantTS$ and *currentC* for each as per write phase 2 processing.

6. With all operations executed and contention resolved, the replica clears $conflictC$ and replies to the $w1$ message in the RESOLVE request that caused it to freeze, if there was one. The reply to the RESOLVE request is generated based on the rules for processing a WRITE-1 request; it will ordinarily be a WRITE-2-ANS.

4.5 Client Protocol

4.5.1 Resolve Responses

The response to a RESOLVE request is processed by the client in the same way as the response to a WRITE-1 request, detailed in Section 3.3.2. Under all normal circumstances the $w1$ request will be executed during resolution, and the response will be a WRITE-2-ANS. Exceptions to this may only occur if the client submits duplicate operations, or an operation that was not part of the contention, in which case the response will be a WRITE-1-OK or WRITE-1-REFUSED. Responses from contention resolution may differ when running the protocol with symmetric key *authenticators* rather than public key signatures in messages, described in detail in Chapter 8.

4.5.2 Write Phase 2 Responses

As mentioned in Section 3.3.2, contention resolution may interfere with the responses to a client's WRITE-2 request. This occurs when a client is executing phase 2 of the write protocol while contention resolution is requested by another client. It is possible for a subset of replicas to accept the WRITE-2 request before resolution begins, but have the client's operation reordered as part of the contention resolution process. In this scenario the client may receive some WRITE-2-ANS responses for its originally assigned timestamp, and later receive others for the new timestamp and higher viewstamp after resolution. If the client receives a quorum of matching responses it is done; otherwise it retries the WRITE-2 with the highest certificate received, soliciting a quorum of matching responses.

It is important to note that if a client ever receives a quorum of matching WRITE-2-ANS responses, this operation has committed and will never be reordered at the replicas. This is ensured by the quorum intersection property—if a client receives $2f + 1$ WRITE-2-ANS responses for a given timestamp, $f + 1$ of these must be from non-faulty replicas, and at least one of these must be a participant in contention resolution, providing knowledge of the committed operation.

4.6 Impact on HQ Operation

Contention resolution affects the operation of the HQ protocol, in that multiple certificates may exist with the same timestamp but different viewstamps. This possibility has no effect on WRITE-1 or READ processing, but does have an impact on WRITE-2, RESOLVE and writeback messages.

If a replica receives a client request containing a valid certificate for the current timestamp, but a higher viewstamp, the replica's current tentative state must have been superseded as a result of contention resolution. The replica rolls back its current state by replacing *currentC* with *backupC* and replacing the most recently modified entry in *oldOps* with the operation stored in *prevOp*. The replica then processes the request received in the client's message, updating *vs*. If the request is a WRITE-2 or writeback, and the replica does not have a copy of the corresponding operation, it obtains the operation via state transfer as described in Chapter 6.

If a replica receives a client request with a certificate for *currentC.ts*+1, but with a viewstamp greater than $currentC.vs$, then contention resolution has occurred between timestamps $currentC.ts$ and $currentC.ts + 1$. Since the operation for $currentC$ may have been rolled back during resolution, the replica replaces $currentC$ with $backupC$ and reverts state as above, then requests state transfer between $currentC.ts$ and $currentC.ts + 1$. It is safe to roll back $currentC$, since the $f + 1$ valid replicas that participated in BFT for the new viewstamp must have a record of the operation committed at $currentC.ts$.

4.7 View Changes

The primary replica of the BFT module is responsible for collecting START messages into a valid *startQ* structure, and initiating a round of BFT to establish consistent knowledge of contending state. The failure of the primary may halt contention resolution, and prevent system progress. We thus need a mechanism to change the primary when such failures are detected.

The BFT protocol, as influenced by Viewstamped Replication [34], uses a view change mechanism to change primaries. A simplified overview of the view change protocol is presented as follows, with full details in [7]:

1. A replica sets a timer when it receives a client request that has not yet been executed. Clients initially send requests only to the primary, but will broadcast a write request to all replicas (who then forward the request to the primary) after their own client timeout.

2. If the timer expires before the operation is executed, the replica freezes and sends a VIEW-CHANGE message to all other replicas. The VIEW-CHANGE message nominates the next primary p, which is chosen in round-robin order via $p = v \mod R$, for view v and replica set size R.

3. The new primary collects $2f + 1$ VIEW-CHANGE messages and sends a NEW-VIEW message to all replicas, establishing itself as the new primary.

We build upon the BFT view change protocol to change primaries when failures are encountered, providing both safety and liveness. There are two situations where the primary may fail or misbehave: failing to correctly initiate the BFT protocol after receiving a quorum of START messages, or sending an invalid *startQ* as the BFT operation.

4.7.1 Failure to initiate BFT

A BFT replica only requires one timeout to initiate a view change. Once a replica forwards a request to the primary, it can assume that it is the primary's responsibility to start the agreement process, and ensure the subsequent execution of the operation. If this does not occur before the timeout, increased exponentially each view change to prevent spurious view changes due to network delays, it is safe to perform a view change. This is not the case in the HQ protocol, as addressed below.

The HQ module at the primary is unable to build a *startQ* structure and initiate BFT until it receives a *quorum* of START requests from replicas. A bad client may send a RESOLVE message only to a single replica, halting progress regardless of the primary. We thus need an additional timeout before the BFT view change timer is started, this first timeout triggering the replica to broadcast the RESOLVE to all others. Once all replicas have seen the RESOLVE, the BFT view change timer can be started, since these replicas will send a quorum of START messages to the primary. Since a replica always broadcasts the RESOLVE message well before sending a VIEW-CHANGE, we guarantee that contention will be resolved (likely following a system view change), ensuring no HQ module is left frozen.

The HQ view change protocol is as follows:

1. A replica starts a *broadcast* timer after it receives a RESOLVE request from a client and has sent a START message to the primary.

2. The *broadcast* timer stops when a BFT PRE-PREPARE message is received corresponding to the current contending timestamp. The BFT module is required

to notify the HQ proxy of the receipt of a PRE-PREPARE. At this point the BFT *view change* timer will start as part of the standard BFT protocol.

3. If the *broadcast* timer expires, the replica sends the RESOLVE message to all other replicas. It makes an up-call to the BFT module to start the BFT *view change* timer, for the resolve operation corresponding to the current contending timestamp.

4. If a replica receives a new RESOLVE message via another replica rather than a client, it immediately forwards the RESOLVE to all replicas itself, and starts the BFT *view change* timer. Note that the replica must broadcast the RESOLVE itself, since it cannot be sure the sending replica forwarded the RESOLVE to a full quorum of valid replicas.

5. If the *view change* timer expires, the BFT module will broadcast a VIEW-CHANGE message and proceed with the view change protocol. It must also notify the HQ module: If the HQ code has not already done so, it forwards the RESOLVE message to all replicas. This ensures that all replicas are aware of contention resolution in the case where a faulty primary sends PRE-PREPARE messages to less than a quorum of replicas, leading to an insufficient number of replicas requesting a view change[1].

6. BFT will ensure that view changes continue until a valid primary is chosen, and the contention resolution operation is executed. The only additional communication required with the HQ proxy is notification of the new primary and view number after the operation is committed.

[1]This is not a concern in the standard BFT protocol because the frozen replicas will halt progress on any subsequent valid write request, ensuring a quorum of VIEW-CHANGE requests. In HQ however we must ensure that if the HQ module at one replica is frozen due to contention, contention resolution for that timestamp *must* occur, otherwise the frozen replica will halt progress in the HQ quorum protocol, unequipped with a view change mechanism.

Appropriate Timer Values

It is important to maintain appropriate values for the *broadcast* and *view-change* timers, to avoid spurious view changes, and to with minimize communication overhead. The *view change* timer is handled internally by the BFT protocol, and follows the protocol described in [7], doubling the timer each consecutive view change to account for unknown message delays. Liveness is provided assuming message delays do not indefinitely grow faster than the timeout interval.

The *broadcast* timer is less critical; an insufficient duration will cause replicas to unnecessarily broadcast RESOLVE messages, yet will not trigger a premature view change. This timer is set to a multiple of the expected interval between RESOLVE and PRE-PREPARE, measured experimentally with system deployment.

4.7.2 Bad *startQ*

The BFT protocol will not stop its *view change* timer until the current operation has committed; the *operation* in BFT is the *startQ* corresponding to the current contention timestamp. We need to add additional functionality to the protocol to detect if this *startQ* is valid however, and trigger a view change otherwise. This is achieved via a simple extension to the state processing following the round of BFT. All replicas check the validity of *startQ*, and if it does not contain a quorum of valid START messages, make an upcall to BFT to immediately send a VIEW-CHANGE request.

Chapter 5

Optimizations and Extensions

This chapter details a number of optimizations to the HQ protocol, which result in significantly improved performance. We also describe extensions to HQ Replication to support the execution of multi-object transactions.

5.1 Hashed Response

In the unoptimized HQ protocol, each replica responds in full to a successful WRITE-2 or READ request. Under normal circumstances, however, the results in these messages are identical, resulting in the transmission of at least $2f$ redundant copies of the operation result. This is a particular concern when the operation result is large, such as a file read. Instead we have clients identify a designated replica in a WRITE-2 or READ request. The designated replica responds in full, while the others send a hash of the operation result, rather than the result itself. Matching hashes are sufficient to identify a successful operation, and the client can retry with a different designated replica if it doesn't receive a full copy of the result in the first $2f + 1$ responses.

This optimization is only of utility where operation results are of appreciable size. For applications where results are small, the computational overhead of computing result hashes is not justified by the bandwidth savings.

5.2 Optimistic Grants

The HQ protocol supports general operations, which may take significant time to execute at each replica. A simple optimization is to allow replicas to respond to any WRITE-1 request while this execution in process. This is equivalent to a scenario where the replica executed the previous operation extremely quickly, and hence does not compromise correctness. The replica cannot respond to a WRITE-2 request until the execution has completed, however, to ensure a serial execution of operations.

5.3 Delayed Certificates

Certificates are included in responses to WRITE-1 and READ requests to allow clients to perform a writeback to slow replicas. They are also included in WRITE-2-ANS messages in case it differs from the certificate in the WRITE-2; as discussed, this can happen as a result of contention resolution executing a different operation in place of the client's, prompting the client to retry the WRITE-2 with the new certificate. Both scenarios are rare under typical protocol operation, and hence we can optimize for the more common case when these certificates aren't required.

WRITE-2-ANS responses are modified to contain the client *op#* instead of the write certificate, in cases where the certificate in the response would have been the same as that in the WRITE-2 request. The *op#* allows the client to identify responses to previous operations. A certificate is included in the WRITE-2-ANS message if it differs from the original request, owing to contention resolution. It is also included if the WRITE-2-ANS is in response to a WRITE-1 request from the client, in the scenario where a different client completed the operation on its behalf, and the requesting client does not yet know this.

If message loss and replica failure are low, all replicas will remain in relative synchrony, and writebacks to slow replicas will be uncommon. In this typical scenario we may avoid the inclusion of certificates in WRITE-1-OK and WRITE-1-REFUSED messages entirely. Similarly, if it is rare for reads and writes to be issued on the same object concurrently, then READ-ANS messages will match and the certificates may be replaced by a timestamp and viewstamp. If the responses to an optimized request do not match, then the client retries its original operation in the unoptimized form. This will ensure that the client obtains the certificates necessary to perform a writeback.

5.4 Piggybacked Grants

Section 4.4.4 describes an additional phase of communication following BFT in the contention resolution protocol. This phase is used to obtain grants to build a certificate for executing each operation in L, the ordered list of all contending operations. This protocol is described for simplicity, with minimal modifications to the BFT module. We can avoid this additional phase however, if we modify the module protocol to produce HQ grants for each operation while running BFT.

Under the standard BFT protocol, an upcall is made to the application code to execute an operation, once a quorum of valid COMMIT messages is received. In the BFT module used by HQ, this upcall is made to the HQ server code, to communicate consistent state on all contending operations. We now modify this upcall such that it also includes a certificate for each operation.

Prior to sending a COMMIT message while executing BFT, the module performs a MAKE-GRANT(*startQ*, *vs*) upcall to the HQ server proxy code, passing the *startQ* contents of the BFT operation, and the *vs* viewstamp corresponding to the given round of BFT. The HQ code determines the grants that would have been sent in the additional phase of communication from Section 4.4.4, and returns these in its response. BFT then piggybacks these grants on the COMMIT message sent to each other replica.

Once the BFT replica receives a quorum of valid COMMIT messages, it passes the grants, *startQ*, and the current viewstamp up to the HQ server code for state processing. Under normal circumstances these grants will form a valid certificate to execute each operation, and the state processing code won't require an additional phase of message exchange to gather these certificates. If there are malicious replicas however, the grants may not form valid certificates, in which case the message exchange is still required. For simplicity we do not require BFT to verify the grants in COMMIT messages, hence the possible need for this post-phase.

While the grants produced by BFT could be collected by a malicious intruder or bad replica, this does not compromise the safety of the protocol. We cannot rely on the fact that the operation corresponding to each grant will eventually complete, since the operations may be aborted if a view change occurs before $f + 1$ honest replicas send their COMMIT messages. We can however ensure that no replica is able to gather a certificate for any operation that is not eventually executed at the same timestamp, as a result of BFT. If a BFT operation is aborted following a view change, then there must be at most $2f$ copies of COMMIT messages and grants, otherwise the BFT operation would persist throughout the view change. This precludes the existence of a certificate for any client operation that is assigned a grant. A malicious entity is also unable to combine the partial set of grants formed in an aborted view with those from a subsequent view, since these will contain different viewstamps and hence not form a valid certificate.

5.5 Preferred Quorums

During failure-free executions, only $2f + 1$ replicas are required to successfully run the HQ and BFT protocols. This common case can be exploited through the use of *preferred quorums*, introduced in Q/U[1] and similar to the use of witnesses in Harp [21]. A preferred quorum is a set of $2f + 1$ replicas with which each client communicates, avoiding the cost of running the protocol at f additional replicas, and ensuring all client operations intersect at the same set of replicas. The latter feature significantly reduces the frequency of writebacks, since once an operation has committed it is known by all replicas in the preferred quorum, avoiding the occurrence of "slow" replicas. Progress will halt if any replica in the preferred quorum fails, at which point

[1]Note that while HQ and Q/U both support preferred quorums, HQ is able to reduce the number of replicas in active communication from $3f + 1$ to $2f + 1$, while Q/U only from $5f + 1$ to $4f + 1$

another replica needs to be added to the preferred quorum.

The replicas in the current preferred quorum are determined statically, to ensure that all clients communicate with the same preferred quorum. We set the preferred quorum membership to include the current primary, along with the $2f$ subsequent replicas in the ID space.

While only $2f + 1$ replicas are required to perform each write operation, clients send their WRITE-1 request to all replicas, to avoid excessive state transfer during failures. Only the replicas in the current preferred quorum respond to the WRITE-1 request, and only these replicas participate in phase 2 of the write protocol.

If a client does not get a full quorum of valid responses within a timeout, it retries the operation using the normal non-preferred quorum protocol. If the client receives timely responses for a period of time, it can again return to preferred quorum operation.

Replicas outside the preferred quorum need to periodically learn the current system state. This transfer can be performed efficiently, since in the absence of malicious clients the non-preferred replicas already hold a log of outstanding client operations. Periodically, a non-preferred replica will ask those in the preferred quorum for a lightweight state transfer, identical to regular state transfer except that operations themselves are not transmitted. Instead each log entry is transmitted as $< cid, oid, op\#, h_{op} >$, containing an operation hash in place of the operation. The non-preferred replica does not have knowledge of the most recent timestamp, and hence leaves this field empty in the state transfer request, requesting state up to that most recently executed at each replica. If the non-preferred replica does not have an operation in its WRITE-1 log corresponding to a given timestamp and h_{op}, it requests standard state transfer for this interval.

5.6 Transactions

The server application data in HQ is typically partitioned into multiple independent objects, numbered by *oid*, and corresponding to files or logical groupings of data.

This partitioning is employed to minimize write contention—the HQ protocol is run independently for each object, allowing concurrent writes on separate objects. There are clearly scenarios where a transaction is required to run atomically across a number of these objects however; this functionality is provided in HQ by extensions for multi-object transactions.

The correct execution of a transaction requires that it is executed in the same order across all (a quorum of) replicas, for each object. Furthermore, the transaction must not be interleaved with any other transactions. For example, if transactions A and B intersect at two objects $o1$ and $o2$, it must not be the case that operation A_1 is ordered before B_1 on object $o1$ and operation A_2 is ordered *after* B_2 on object $o2$.

To meet the conditions for correct serialization of transactions, we require clients to fetch a grant across all relevant objects in a single request. This is performed using a modified WRITE-1 request of type $\langle$WRITE-1, *cid, oid*$_1$*, op#*$_1$*, ..., oid*$_k$*, op#*$_k$*, op*$\rangle_{\sigma_c}$, containing an *oid* and *op#* for each of the k objects in the transaction. *op* is a multi-part operation, possibly consisting of separate operations $op_1, ..., op_k$ as perceived by the server application code. We term the subsequent collection of grants a *multi-grant*, of the form $\langle$*cid, h, vs, olist*$\rangle_{\sigma_r}$, where *olist* contains an entry $\langle$*oid, op#, ts*$\rangle$ for each object in the request. A replica will only assign a multi-grant to a client if it has *no* outstanding grants for any of the objects in the transaction.

If a replica does have outstanding grants for any of the transaction objects, it will refuse the write request and return a WRITE-1-REFUSED for each outstanding grant. The client can then use these to retry the multi-object transaction.

The requirement for only one outstanding request per object at the client remains, in that the client cannot request any operations on any of the objects in the transaction until it has completed.

To provide support for multi-object transactions, a single instance of BFT is used to service contention resolution across all objects. This is distinct from the protocol previously described where a separate BFT module is kept for each object. The previous description was purely for the purposes of clarity—BFT serializes all operations at the primary, and hence is able to order across multiple objects provided each HQ module has consistent knowledge of the BFT primary.

The protocol for running multi-object transactions is as follows:

1. A client sends a $\langle$WRITE-1, *cid, oid*$_1$*, op#*$_1$*, ..., oid*$_k$*, op#*$_k$*, op*$\rangle_{\sigma_c}$ request to all replicas.

2. If a replica has no outstanding grants for any of the objects $oid_1, ..., oid_k$ it will assign a grant to the client for each, and respond as per a WRITE-1-OK, with multi-grant $\langle$*cid, h, vs, olist*$\rangle_{\sigma_r}$ in place of the regular grant, for *olist* consisting of a $\langle$*oid, op#, ts*$\rangle$ for each object. If the replica has an outstanding grant for one or more objects, it will send a WRITE-1-REFUSED to the client, containing the current grants for any outstanding operations on the objects. In each case the response also includes the most recent certificate for each requested object, rather than the single certificate in a standard WRITE-1-OK or WRITE-1-REFUSED message.

3. If the client receives a quorum of matching multi-grants it may proceed with phase 2, sending a certificate composed of the quorum of multi-grants. Otherwise it performs a WRITEBACKWRITE or requests contention resolution depending on whether it received refusals or mismatched multi-grants. The WRITEBACKWRITE contains a certificate for each refused object, paired with the original WRITE-1 request. A RESOLVE request contains a quorum of conflicting multi-grants.

4. A replica processes a valid WRITE-2 request by making a single upcall to the application. It must first ensure that it is up to date with respect to every object in the transaction, achieved via state transfer if necessary. The single upcall allows the application to run the transaction atomically, in the right place in the order.

5. In response to a RESOLVE request, a replica freezes for *all* objects in the request, and performs contention resolution for them simultaneously. The START message contains information for each object identified in the RESOLVE request. As

there is only a single instance of BFT per replica, across all objects, the BFT agreement protocol executes as per a single-object transaction.

6. *startQ* processing is similar to that for a single-object request. Replicas retain at most one valid request per client *per object*. The grants exchanged with other replicas following the deterministic ordering of requests will comprise of multi-grants if some of the requests are multi-object transactions. The final timestamp for each object o is set to $o.currentC.ts + |L_o|$, where L_o is the subset of L containing all requests that involve object o. The operations are performed in the selected order once the replica receives $2f + 1$ matching multi-grants for each operation.

Any client requesting a single-object operation at a replica that has already issued a multi-grant will receive the outstanding multi-grant in the WRITE-1-REFUSED response. The client will perform a writeback on this multi-grant if it gets a quorum of matching responses for a different client; otherwise it handles the response as it would a regular WRITE-1-REFUSED reply.

While functional, the protocol for implementing multi-object transactions entails significant overhead under high load, owing to the low likelihood that a quorum of matching multi-grants will be received. Agreement-based replication scheme such as BFT do not have this problem since they consider the system as a single logical object.

Chapter 6

State Transfer

This chapter describes the state transfer mechanisms in HQ. This is used to bring slow replicas up to date with the current system state.

Since HQ and BFT require only $2f + 1$ replicas to participate in each round of the protocols, some replicas may drift behind the most recently committed state. Messages are sent to all replicas, and hence under normal circumstances all honest replicas will have knowledge of the most recently agreed-upon operation. However, message loss or malicious behavior can lead to some replicas without this knowledge.

We first describe a simplified state transfer protocol where replicas maintain unbounded logs of all committed operations. This protocol is then extended to use *snapshots* of system state to truncate operation logs. We also describe optimizations to minimize the state transfer overhead using hashed responses, and discuss modifications to remove BFT state transfer from the BFT module, replacing it with more efficient HQ-level state transfer

We discuss state transfer here only in the context of public key cryptography; modifications to support the use of symmetric key cryptography are given in Section 8.4.

6.1 Simple State Transfer

Replicas keep a log of committed operations to facilitate state transfer. This log is of the form $\{< cid, oid, op\#, op >, ...\}$, and contains each committed operation, along with the identifying information contained in the write certificate for the operation. Entries are added to this log once a certificate for a subsequent operation is received (hence the most recent log entry corresponds to *backupC*), or when operations are executed as a result of contention resolution. We assume in this section that there is no bound on the length of this log; Section 6.4 describes modifications to the state transfer protocol to allow the log to be truncated.

Slow replicas request state transfer using a TRANSFER request, specifying the timestamp range for the state that is required. This range extends from *currentC.ts*, the timestamp for the most recently executed operation, to *newC.ts*, the timestamp for the most recently received certificate. State for *currentC.ts* is required even though an operation has been executed at that timestamp, since this operation is only tentative and may not have committed. Replicas respond to a state transfer request with a STATE message, containing log entries covering the requested range; these log entries may extend further than the range if the responding replicas have progressed beyond *newC.ts*. We describe this state transfer protocol in more detail below.

6.1.1 "Client" Protocol

We first describe the state transfer protocol from the perspective of the "client", where the client of state transfer is an HQ replica that is behind with respect to the global system state.

1. State transfer is triggered when a replica receives a valid write certificate *newC* with $newC.ts > currentC.ts + 1$, or if $newC.ts = currentC.ts + 1$ and the replica has no operation corresponding to *newC.h* in *ops*.

2. The replica forms a request $\langle \text{TRANSFER}, currentC.ts, newC.ts \rangle_{\sigma_r}$ and sends

it to the $2f + 1$ replicas that signed *newC*, requesting state information for all operations from *currentC.ts* to at least *newC.ts*. State for *currentC.ts* is requested since it is only tentative at the the slow replica.

3. The replica waits for $f+1$ matching valid responses. Matching responses contain identical log segments from *currentC.ts* to $newC.ts - 1$; a matching *currentC* certificate for $newC.ts - 1$ may be substituted for the final log entry, if a responding replica has not yet received a certificate for *newC.ts*. Valid responses must also contain valid certificates, have *backupC* in the response match the highest log entry, and have an entry in *ops* corresponding to *currentC.h* in the response.

4. Matching log entries are replayed in order, with the replica executing each operation, and updating its protocol state accordingly. These matching entries may extend beyond *newC.ts*, if system state has advanced since *newC.ts* was received.

   The replica may have to undo the operation at *currentC*, before executing the log, if the log entries for *currentC.ts* don't match the most recently executed tentative operation.

   The matching log entries may also extend only to $newC.ts - 1$, if the operation at *newC.ts* is not yet committed; in this case the replica waits for a response that contains the operation for *newC.h* in *ops*, and executes this operation.

5. *currentC* and *backupC* are replaced by the certificates corresponding to the two most recently executed operations. *backupC* will be included in the response from the replica r with the shortest log sequence, either as *r.backupC*, or *r.currentC* if the replica had not yet seen the certificate for *newC.ts*.

   currentC is replaced with either *r.currentC*, or *newC* if this operation was executed based on a response in *ops*.

6. If a response $r2$ exists such that $r2.currentC.ts > currentC.ts$, for the new *currentC* at the replica, then the replica is not yet up to date with the most

recent system state. It requests state transfer again between *currentC.ts* and *r2.currentC.ts*.

A complicating factor in the processing of replica state during state transfer is that no certificates are included in the log entries; certificates are avoided for efficiency reasons, both in bandwidth and log storage. A result of this is that replicas are unable to update the write certificate field in *oldOps* for any clients with operations in the log. We note that this is in fact unnecessary, since the certificates in *oldOps* are used only to respond to duplicate WRITE-2 requests, and the certificate in such a response is unnecessary since the operation must have already committed.

When updating *oldOps* while replaying the log, the certificate for any modified entry is set to *null*. If responding to a duplicate client request, the replica sends its WRITE-2-ANS response with this *null* certificate. $f+1$ matching responses with *null* certificates is sufficient proof to a client that the operation committed with the given result. Any replica with *currentC.ts* greater than the timestamp for the operation is able to send a *null* certificate, confirming that the operation is no longer tentative.

6.1.2 Replica Protocol

The protocol for responding to a state transfer request at a replica is straightforward, and operates as follows:

1. A replica drops any $\langle$TRANSFER, $lowerTS$, $upperTS\rangle_{\sigma_r}$ message if $upperTS >$ *currentC.ts*+1, i.e., if the replica does not have sufficient information to respond to the request. It also ignores requests that are incorrectly signed.

2. The replica responds with a $\langle$STATE, $< cid, oid, op\#, op >_{lowerTS}$, ..., $< cid, oid, op\#, op >_{backupC.ts}$, *backupC*, *currentC*, *ops*$\rangle$ message, i.e., it sends its entire log history from $lowerTS$ onwards, the current and backup commit certificates, and all operations in *ops*.

6.2 Hashed Responses

In the protocol thus described, each replica that receives a state transfer request must send a complete copy of all requested log entries. This can be expensive when the slow replica is significantly behind the latest system state, or when operations are large.

Instead we adopt a more bandwidth-efficient approach, where only one replica sends a complete response, while the others send hashes. We modify the TRANSFER request to contain an additional *dr* field, indicating the *rid* of the *designated replica*. The designated replica is responsible for returning a full response to the request, while the other replicas replace the log segment up to $newC.ts - 1$ with a single hash over all entries (including an entry for $newC.ts - 1$ based on *currentC*, if $currentC.ts = newC.ts - 1$). This hash excludes the operation corresponding to *newC.ts* since it may not yet have been committed, as well as any possible operations with higher timestamps, since replicas may have log segments of different length when receiving the TRANSFER request. The log entries from *newC.ts* onward are sent by all replicas, along with the certificates *backupC* and *currentC*, as well as the the operation corresponding to *currentC*.

If f hashes match the log segment from the designated replica, then the optimization was successful and the replica can continue processing of the responses. Otherwise it retries the state transfer request in the unoptimized form, and will choose a different designated replica the next time state transfer is required.

6.3 Checkpoints in BFT

We discuss the truncation of operation logs in Section 6.4; we first describe the use of *checkpoints* in BFT, however, to provide a context for the modifications to state transfer.

The traditional BFT protocol performs state transfer through the exchange of prepare certificate logs. These logs cannot be allowed to grow indefinitely, however, and must eventually be truncated. The pruning of these logs is performed by the BFT *checkpoint* protocol, one of the more complex and expensive aspects of BFT. During a checkpoint, replicas compute a digest over the entire application state, store a logical copy of this state, and then exchange signed copies of these digests with other replicas. Once a replica holds $f + 1$ signatures for the same digest, it has proof that the state is correct and known at at least one legitimate replica, and can hence prune any prior log entries.

While the computation of digests is expensive, checkpoints are a necessary aspect of the standard BFT protocol and must occur frequently, particularly to support view changes. On a view change the new primary must establish the most recent stable system state, and run the BFT protocol for operations that may have committed since then. The new primary is unable to compute a stable system state based on prepare logs alone, since operations may commit out of order, and bad replicas may send conflicting prepare messages for old operations in previous views. A checkpoint certifies that the state is indeed stable at at least one valid replica. The digest is required so that state transfer is possible from a single valid replica, without requiring f other corroborating copies of the state.

It would be highly undesirable to run the full checkpoint protocol at both the BFT and HQ layers in the HQ protocol. Fortunately this is unnecessary, since BFT is used by HQ in a far more restricted context than the original protocol. Moreover, BFT can rely on HQ to perform state transfer for operations ordered in any committed contention resolution operation.

6.3.1 Removing Checkpoints from BFT

The BFT module described thus far has closely followed the original BFT specification [7]. We have discussed the use of BFT in a relatively unmodified fashion for the purposes of clarity, yet the expense involved in the original BFT state transfer protocol warrants modifications when deployed in HQ.

We are able run the BFT module with a significantly stripped-down state management protocol, owing to fundamental differences between our BFT module and the stand-alone BFT protocol. The BFT module as used in HQ differs from the standard BFT protocol in two key ways:

1. The BFT module has no application state. BFT is used by HQ purely to order contention resolution operations, with application state maintained by HQ itself. We can leverage HQ state transfer to bring slow replicas up to date, and are able to avoid the inclusion of digests in checkpoints.

2. Unlike standard BFT, the HQ BFT module has no request *pipeline*—replicas freeze when requesting contention resolution, and will not issue a subsequent request until the previous one has been resolved. We can derive from this a guarantee that when BFT runs an operation, the state from any previous BFT operation will be known at at least $f+1$ valid replicas. With knowledge of prior state at $f+1$ non-faulty replicas we are able to remove BFT checkpoints and logs entirely.

6.3.2 Reliance on HQ for State Management

We formalize the previous point 2 in the following invariant:

Lemma 6.3.1 *If a round of BFT is commenced for contention at timestamp t, the application state corresponding to the operation at timestamp $t-1$ must be known at a minimum of $f+1$ non-faulty replicas.*

An informal proof of the above invariant is as follows:

Proof BFT is used by HQ solely for resolving instances of write contention. Instigation of contention resolution requires a conflict certificate containing $2f+1$ grants with matching timestamps; $f+1$ of these must have come from honest replicas that have executed the operation corresponding to the previous timestamp. There are three possible scenarios by which the previous timestamp was agreed upon:

1. The previous timestamp was established via the standard HQ protocol. The quorum property of HQ guarantees that at least $f+1$ honest replicas know about the operation.

2. The previous timestamp was established using contention resolution, which ran until completion. Here BFT guarantees that $f + 1$ honest replicas know about the operation.

3. The previous timestamp was established using contention resolution, but the contention resolution process did not complete at all replicas. It is still possible for $2f + 1$ non-frozen replicas to see a certificate for this timestamp—if at least one honest replica commits the operation in BFT, and up to f faulty replicas also commit, the combined $f + 1$ replicas can transfer their state to f slow replicas, resulting in $2f + 1$ non-frozen replicas with matching timestamps. The state transfered to the slow replicas must be valid however, since one honest replica participated. Hence we are again guaranteed at least $f + 1$ non-faulty replicas with knowledge of the operation.

6.3.3 Simplified BFT Protocol

Since $f + 1$ valid copies of system state exist for any previous BFT operation, a replica may participate in a round of BFT for any new timestamp in the current or a subsequent view. It will accept a valid PRE-PREPARE message and send a PREPARE regardless of whether it has committed previous operations. Unlike traditional BFT where a BFT replica must execute each operation in order, the BFT module is able to rely on HQ state transfer to fill in any gaps in the execution. The existence of a quorum of signed START messages in the *startQ* of a BFT request implies that the state for any previous operation can be retrieved on the HQ level.

The standard BFT protocol also maintains a *high water mark* H, used to limit the possible sequence numbers the primary can assign to a BFT operation. The high water mark is included to prevent a bad primary from exhausting the sequence number space. H is not required in the BFT module used by HQ, since the primary is constrained to assign a sequence number one higher than that in the viewstamp for the certificates in *startQ*. To provide greater separation between the BFT and HQ modules, and avoid performing an upcall to HQ to validate the *startQ* and sequence number in the PRE-PREPARE phase, we instead provide proof of sequence number validity on the BFT level. This proof is composed of $2f + 1$ signed COMMIT messages from the previous BFT operation. Replicas will only accept a PRE-PREPARE message for a sequence number s if it contains $2f + 1$ signed COMMITs for sequence number $s - 1$. The primary may retrieve copy of these COMMIT messages from any valid replica that committed the operation, if it did not receive $2f$ COMMIT messages while running BFT.

Checkpoints aren't required in the BFT module, and we replace the log with a copy of the operation from the most recent round of BFT, along with the most recent prepare certificate. Replicas send this prepare certificate to the new primary on a view change; the primary will send a PRE-PREPARE for the most recent of these prepare certificates. The new primary sends no other PRE-PREPAREs in its *new-view* message, since there is no request pipeline.

A replica may receive a prepare or commit certificate yet not have seen the original operation. It asks each replica included in the certificate for a copy of the operation; it will receive the operation if the certificate is current, or a refusal otherwise. If the replica receives $f + 1$ refusals, it aborts the BFT round and becomes unfrozen - a refusal from an honest replica implies that it holds a more recent BFT operation, further implying that the old operation is known by at least $f + 1$ honest replicas.

6.4 State Snapshots

We now modify the HQ state transfer protocol to allow replicas to truncate their operation logs. This is a necessary modification to support the execution of long-lived systems.

We restrict the operation log to a maximum length max_{log}, established statically at all replicas. The log is stored in a circular structure, and once it reaches this maximum size, the oldest log entry is overwritten with each new addition. State transfer requests are usually served out of this log; however a particularly slow replica may require information from old log entries that have been overwritten. To facilitate state transfer in this scenario, replicas send application level state corresponding to a fixed snapshot of the service, along with any log entries since this snapshot. We adopt the terminology *snapshot* instead of *checkpoint*, since HQ snapshots occur far less frequently than BFT checkpoints, and do not require inter-replica exchange of signed digests. Snapshots also include a copy of *oldOps*, to capture protocol state along with the application state. Figure 6-1 illustrates the circular log used for state transfer, along with the snapshot state.

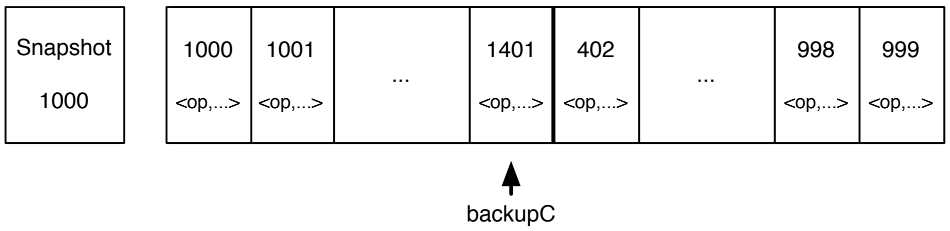

Figure 6-1: Circular log structure used to store state history. In this diagram $max_{log} = 1000$, $backupC.ts = 1401$, the last snapshot was performed at timestamp 1000, and log entries are stored from timestamps 402–1401. When the next operation commits, the entry for timestamp 402 will be replaced with timestamp 1402.

After every max_{log} committed operations, a replica stores a copy of the current committed application state and *oldOps* as a snapshot, and discards the previous snapshot. By matching the checkpoint interval to the length of the log, we ensure that no log entries are discarded for operations that aren't reflected in a snapshot, while performing application snapshots as infrequently as possible. Note that snapshots do not capture the tentative state reflected by executing the operation in *currentC*, but rather only that reflected by *backupC*, since the former may be rolled back during contention resolution. All valid replicas perform snapshots at the same timestamp, and thus at least $f + 1$ honest replicas will have matching snapshots at timestamps after the snapshot transition.

It may be possible for fewer than $f + 1$ matching snapshots to exist during a snapshot transition, where some of the replicas have performed their new snapshot and some have yet to do so. This does not pose a liveness problem since the replicas yet to perform the snapshot are able to do so without requiring state transfer themselves, once they receive the certificate for the most recent operation. Any slow replica requiring state transfer during this transition may safely retry, adopting the policy described in the following section.

6.4.1 Optimistic State Transfer

The signed checkpoint digests in traditional BFT allow state transfer from a single replica. We avoid these signed digests in HQ, but instead rely on an *optimistic* protocol for state transfer, requiring matching snapshots and log entries from $f+1$ replicas. We guarantee that state transfer will succeed if the system is stalled, as required for liveness, and also that it will complete successfully under realistic network delays, but do not guarantee that state transfer will succeed otherwise. This optimism does not compromise system correctness or liveness, but is discussed here for completeness.

The snapshot interval is large with respect to the distribution of round-trip times on the Internet. Hence with very high probability all honest and up-to-date replicas will receive a state transfer request within the same snapshot period. The recipient replicas will respond to the slow replica, and deliver at least $f+1$ matching state transfer responses. This will also be the case when the system has stalled and is waiting for the slow replica to catch up, since no operations will be processed in the meantime. If the system is not stalled and network delays vary wildly, however, the honest replicas may receive state transfer requests in different snapshot intervals, resulting in fewer than $f+1$ matching responses.

Our approach is simply to have a slow replica retry a state transfer request when if hasn't received $f+1$ matching responses after a timeout. The retry is sent to all replicas, in case all replicas have moved to a newer snapshot in the intervening period. The timeout is set experimentally, originally to twice the maximum observed request response time, and doubled on each subsequent failed transfer request as per the view change timer described in Section 4.7.

6.4.2 Replica Protocol

The protocol for handling a state transfer request using snapshots is as follows:

1. A replica ignores any $\langle$TRANSFER, $lowerTS$, $upperTS$, $dr\rangle_{\sigma_r}$ message if $upperTS > currentC.ts{+}1$, i.e., if the replica does not have sufficient information to respond to the request. It also ignores requests that are incorrectly signed.

2. If an entry for $lowerTS$ is still contained in the commit log, the request can be served without the use of snapshots. The replica responds as described in Section 6.1, including the optimizations in Section 6.2.

3. If no entry for $lowerTS$ exists in the log, then some of the required state has been truncated. The designated replica sends the snapshot state[1], and any subsequent log entries up to $upperTS$ in a $\langle$STATE, $snapshot$, $< cid, oid, op\#, op >_{snapshotTS+1}$, ..., $< cid, oid, op\#, op >_{backupC.ts}$, $backupC$, $currentC$, $ops\rangle_{\sigma_r}$ message. Non-designated replicas reply with $\langle$STATE, $h_{snapshot}$, h_{range}, $backupC$, $currentC$, $ops\rangle_{\sigma_r}$, including a hash over the log range as described in Section 6.1.

6.4.3 "Client" Protocol

Again we describe the protocol for the slow replica in state transfer.

1. As before, state transfer is triggered when a replica receives a valid write certificate $newC$ with $newC.ts > currentC.ts + 1$, or if $newC.ts = currentC.ts + 1$ and the replica has no operation corresponding to $newC.h$ in ops.

2. The replica forms a request $\langle$TRANSFER, $currentC.ts$, $newC.ts$, $dr\rangle_{\sigma_r}$ and sends it to the $2f + 1$ replicas that signed $newC$, requesting state information for all operations from $currentC.ts$ to at least $newC.ts$. The designated replica is specified in dr.

[1]It is not necessary to send the complete system snapshot, but rather only the pages that have been modified since the last operation known at the slow replica. This optimization is made possible through the comparison of hierarchical state digests, similar to Merkle trees, as described in [6]

3. The replica waits for $f+1$ matching valid responses. Matching responses consist of a valid response from the designated replica, along with f other responses with hashes matching the snapshot in the designated response (if the response contains a snapshot), and with hashes matching the designated replica's log segment up to $newC.ts - 1$. As in Section 6.1, the *currentC* response from a replica may be substituted for its log entry at $newC.ts - 1$, if the replica has not yet seen a certificate for *newC.ts*. The *backupC* and *currentC* responses from each replica must be valid, and an operation for *currentC.h* must exist in the *ops* response.

   Any log entries above $newC.ts-1$ must match up until the shortest log segment; this ensures that there is a *backupC* and *currentC* corresponding to the end of the matching log segment. If the replica does not receive enough matching responses within a timeout, it retries the TRANSFER request with a different designated replica.

4. If the responses contain a snapshot, this is used to replace the application state at the replica, and becomes the new replica snapshot. *oldOps* is also replaced with the copy stored in the snapshot. Matching log entries are then replayed in order, and state updated as described in Step 4 of the simplified client protocol.

5. *backupC* and *currentC* are updated according to Step 5 of the simplified client protocol, along with execution of the operation corresponding to *currentC*.

6. As in Step 6 of the simplified protocol, if *currentC* does not represent the most recent certificate included in the responses, state transfer is requested again for the highest timestamp received.

Chapter 7

Correctness

This chapter presents high-level safety and liveness arguments for the HQ protocol. We expand on these properties in Section 8.3, when discussing the use of symmetric key cryptography.

7.1 Safety

We examine safety specifically in the context of linearizability [12] of system updates; we show that the system behaves like a centralized implementation, executing operations atomically one at a time.

To prove linearizability we need to show that there exists a sequential history that looks the same to correct processes as the system history. The sequential history must preserve certain ordering constraints: if an operation precedes another operation in the system history, then the precedence must also hold in the sequential history.

We construct this sequential history by ordering all writes by the timestamp assigned to them, putting each read after the write whose value it returns.

To construct this history, we must ensure that different writes are assigned unique timestamps. The HQ protocol achieves this through its two-phase process — writes must first retrieve a quorum of grants for the same timestamp to proceed to phase 2, with any two quorums intersecting at at least one non-faulty replica. In the absence of contention, non-faulty replicas do not grant the same timestamp to different updates, nor do they grant multiple timestamps to the same update.

To see preservation of the required ordering constraints, consider the quorum accessed in a READ or WRITE-1 operation. This quorum intersects with the most recently completed write operation at at least one non-faulty replica. At least one member of the quorum must have *currentC* reflecting this previous write, and hence no complete quorum of responses can be formed for a state previous to this operation. Since a read writes back any pending write to a quorum of processes, any subsequent read will return this or a later timestamp.

We must also ensure that our ordering constraints are preserved in the presence of contention, during and following BFT invocations. This is provided by two guarantees:

- Any operation that has received $2f+1$ matching WRITE-2-ANS responses prior to the onset of contention resolution is guaranteed to retain its timestamp t. This follows because at least one non-faulty replica that contributes to *startQ* will have a *currentC* such that $currentC.ts \geq t$. Furthermore, contention resolution leaves unchanged the order of all operations with timestamps less than or equal to the latest certificate in *startQ*.
- No operation assigned a subsequent ordering in the same round of contention resolution can have a quorum of $2f+1$ existing WRITE-2-ANS responses. This follows from the above, since any such operation will be represented by a *currentC* in *startQ*, and retain its original committed timestamp.

A client may receive up to $2f$ matching WRITE-2-ANS responses for a given certificate, yet have its operation reordered and committed at a later timestamp. Here it will be unable to complete a quorum of responses to this original timestamp, but rather will see its operation as committed later in the ordering after it redoes its write-2 phase using the later certificate and receives a quorum of WRITE-2-ANS responses.

The argument for safety (and also the argument for liveness given below) does not depend on the behavior of clients. This implies that the HQ protocol tolerates Byzantine-faulty clients, in the sense that they cannot interfere with the correctness of the protocol.

7.2 Liveness

This section presents the argument that our approach is live. Liveness is presented from the perspective of the replicated service; while we guarantee that the system is live, we do not guarantee liveness for individual clients—this is discussed in Section 7.3.

We assume that if a client keeps retransmitting a message to a correct server, the reply to that message will eventually be received; we also assume the conditions for liveness of the BFT algorithm are met [7].

The argument for liveness is as follows:

- When there is no contention, client write requests execute in two phases (read requests require only a single phase). Each request for a given phase from a correct client is well-formed, and, by construction of the HQ protocol, is replied to immediately by correct replicas (unless the replica is frozen, which we discuss below). Thus, the client eventually assembles a quorum of replies, which allows it to move to the next phase or conclude the operation.

- If some replicas are behind the latest system state, and return grants for mismatching timestamps, they will be brought up to date by a single writeback phase. This single writeback phase is sufficient to bring all replicas to the latest state for the current timestamp, and hence does not impact liveness.

- If a client fails before completing the second phase of a write operation, this write will be executed on its behalf in a single phase by the next client to attempt a write, via a writeback operation.

- When a client needs to resolve contention to make progress, the contention resolution process will eventually conclude, since it consists of executing a BFT operation, and, given the liveness properties of BFT, eventually the operation is executed at good replicas.

- If a client request did not get a reply because the replica was frozen, then the replica was executing a BFT operation to resolve contention. Using the previous argument, eventually the operation is executed at good replicas, leading to the replica unfreezing, and all pending requests being answered.

7.3 Starvation

While the HQ Replication protocol provides liveness for the system as a whole, it does not offer any guarantees on *fairness* for individual clients. It is possible under pathological circumstances for a given writer to be infinitely bypassed by competing clients. This is a characteristic of quorum-based approaches, and a weakness in comparison to primary-driven agreement protocols such as BFT. In practice we ensure that competing clients are able to execute operations whenever contention is detected and resolved using the BFT module.

Chapter 8

Symmetric Key Protocol

This section modifies the HQ Replication protocol to replace public-key signatures with *authenticators* [7]. An authenticator is a vector of Message Authentication Codes (MACs), each using a secret key to authenticate the message for a given replica. We essentially emulate a single one-to-all signature with a set of one-to-one MACs, one for each replica in the system. Authenticators are advantageous since they require much lower CPU overhead than public key signatures; while improvements have been made in the performance of public key signature algorithms, signatures are still more than two orders of magnitude slower to compute than a MAC.

A key disadvantage of authenticators however is that they don't provide the same security properties as signatures. A signature accepted as valid by any good client or replica will be accepted as valid by any other good client or replica; authenticators do not have this property. A given authenticator may contain valid MACs for some replicas in the system, while others are invalid. Moreover, a client cannot ascertain the validity of an authenticator, since it only contains MACs for replicas.

Authenticators and MACs are sufficient in the standard HQ protocol in the absence of failures and contention. If these conditions do not hold however, we are unable to provide any service guarantees. An example where the standard HQ protocol cannot function with authenticators is in contention resolution. Here replicas must determine the latest system certificate based on the messages in *startQ*. We require this to be a deterministic computation, yet with authenticators cannot ensure that good replicas will agree on the validity of each certificate.

We modify the HQ protocol to accommodate the relaxed guarantees of authenticators, as presented below.

8.1 Non-Contention Authenticator Protocol

Authenticators are sufficient in the unmodified protocol for READ, READ-ANS, WRITE-1, WRITE-1-OK and WRITE-1-REFUSED messages. We simply replace the signature in each grant with an authenticator, and likewise with certificates. Clients replace their signature in each WRITE-1 message with a MAC for the recipient replica, and READ requests remain unchanged. READ-ANS, WRITE-1-OK and WRITE-1-REFUSED messages now contain grants and certificates authenticated for each replica in the system, but not the clients themselves, and hence the client is unable to verify the integrity of these elements. Hence we remove the client grant and certificate integrity checks altogether, but must now modify WRITE-2 and writeback processing to account for situations where the client inadvertently provides a bad certificate.

The previous response to a bad certificate in a WRITE-2, WRITEBACKREAD or WRITEBACKWRITE message was to drop the message, since it was an indicator of a faulty client. With this no longer the case, the replica must notify the client of the problem. The replica signs the bad certificate with its own signature, and responds with a WRITE-2-REFUSED message, containing the signed certificate. The client may then request contention resolution using this bad certificate as the conflict certificate *conflictC* in a RESOLVE message. It will be handled as per regular contention by the replicas, which will successfully execute the operation and any other concurrent operations.

We note that this behavior allows a malicious client or replica to continuously instigate contention resolution. This contention resolution, in the absence of write contention, will result in the execution of at least one operation, and hence liveness is not compromised; however it can force a slower path through the protocol.

8.2 Contention Resolution with Authenticators

The main challenge in performing contention resolution with authenticators is determining the latest valid certificate in *startQ*. System correctness requires this decision to be deterministic, yet a seemingly valid certificate at one replica may appear invalid at another. There may also be multiple conflicting certificates for the same latest timestamp but different requests; replicas must correctly determine which one is valid. Such determinations are easy when using signatures, since a valid signature is seen as valid by all replicas, but not so with authenticators. These problems are solved with additional processing before running BFT, including a voting phase where replicas determine the validity of certificates.

Note that communication within the contention resolution protocol itself still uses signatures. This is less of a computational expense since we expect write contention to occur less frequently than regular operations.

The contention resolution protocol begins when one or more replicas receives a valid RESOLVE request. As discussed in Section 4.4.2, a replica will first perform state transfer if it receives a RESOLVE request with $clientConflictC.ts > currentC.ts + 1$ or $clientConflictC.vs > currentC.vs$. Each replica then updates $conflictC$, adds $w1$ to *ops*, and sends a $\langle$START, *conflictC, ops, currentC, grantTS, vs*$\rangle_{\sigma_r}$ message to the primary. They will broadcast the resolve request to all replicas if they don't receive a corresponding response within a timeout, as described in Section 4.4.2. Here *vs* is the current viewstamp at the replica, recorded as another component of the replica state; we need this additional argument when using authenticators because it is possible that the viewstamp in *currentC* is smaller than the most recent viewstamp known at the replica. The viewstamp was not necessary when using signatures, since at least one request will always be executed at a new viewstamp as part of conflict resolution—this may not be the case when using authenticators.

The primary collects these START messages and checks them for validity. In particular, it discards any message containing a *currentC* or $grantTS$ whose request is not present in *ops*; the presence of an operation in *ops* is later used to check for a requests validity. The primary then processes these messages before running BFT, as discussed below.

8.2.1 Step 1

The first problem we need to solve is that it is possible for START messages from different replicas to propose current certificates for the same timestamp but different requests. We say that such proposals *conflict*. Conflicts can happen in two ways:

The first is when some replica is behind with respect to the running of BFT. In this case the viewstamp in its *start.currentC* will be out of date; the primary discards such a START message, since the instance of contention it refers to must already have been resolved. The primary tells the replica to unfreeze via a $\langle$START-REFUSED, *currentC*$\rangle$ message; this will contain at least $f + 1$ valid authenticators for the replica, providing sufficient proof to unfreeze and instigate state transfer to move to the new viewstamp. The replica will send a START message to the primary for the current round of contention when it receives a RESOLVE for the new viewstamp.

The second case occurs when one of the replicas is lying, and pretends that a certificate it holds is valid. This case cannot be handled by the standard protocol, and requires some modifications.

As in our base protocol, the primary collects all START messages into *startQ*, but it also collects non-conflicting messages into a subset $startQ_{sub}$. Each newly received message is checked to see if it conflicts with some message already in $startQ_{sub}$. If there is no conflict, it adds the new message to $startQ_{sub}$. Otherwise the new message is left out, and the existing conflicting message is also removed; if the new message conflicts with several messages already in $startQ_{sub}$, one of them is selected for removal deterministically, e.g., based on replica id. This process is guaranteed to remove at least one START message from a faulty replica.

Step 1 terminates when $|startQ_{sub}| + k = 2f + 1$, where $k = |startQ - startQ_{sub}|/2$ is the number of conflicting pairs. It is safe to wait until this point since each pair contains at least one faulty replica, and hence a maximum of $2f + 1$ responses from honest replicas are required. After termination, $startQ_{sub}$ contains at least $f + 1$ entries from honest replicas, and there are no conflicting proposals for current certificates among the entries in $startQ_{sub}$.

8.2.2 Step 2

The primary now has a collection of at least $2f + 1$ START messages in $startQ$. If the latest certificate in $startQ$ is the certificate formed by each $grantTS$ in the set, or is proposed by at least $f + 1$ replicas, the primary can simply run BFT as discussed in Chapter 4. Otherwise it isn't clear which certificate to identify as the most recent, because it's possible that the certificate with the highest timestamp was proposed by a liar. To solve this problem, the replicas carry out the following protocol.

1. The primary p sends a $\langle \text{CHECK}, startQ\rangle_{\sigma_p}$ message to all the replicas. This message contains the START messages in the order the primary processed them and thus each replica will be able to compute $startQ_{sub}$ using the same computation as the primary.

2. Each replica computes $startQ_{sub}$. It then selects all the certificates C in $startQ_{sub}$ that have timestamps in the range $[currentC.ts - 1, currentC.ts + 1]$, if any exist. This range encompasses the cases where $currentC$ is behind the latest system state, current, or ahead with an inconsistent state, with the following additional constraints:

    - If $C.ts = backupC.ts$, then $C.h = backupC.h$, i.e., both certificates identify the same request.
    - Similarly, if $C.ts = currentC.ts$ then $C.h = currentC.h$.
    - If $C.ts = currentC.ts + 1$, then either C's authenticator appears valid to the replica, or the replica has seen the request represented by C in the latest round of WRITE-1 requests. This latest set of write requests is given by $ops - \{curr\}$, where $curr$ is the write request that was last executed to bring the replica to the current state. It is important to ignore $curr$, since we are only interested in the new WRITE-1 requests in ops.

At this point the replica has selected between zero and three *candidate* certificates.

3. Each replica creates *votes* for all its candidates. A vote is a pair $\langle ts, h\rangle$ where h is the hash of the request at timestamp $ts - 1$. The hash ensures that two votes for a certificate, as represented by the timestamp, are compatible only if the replicas that voted were at the same previous state; in essence preventing a fork of the system state [30]. Note that the inclusion of the hash requires each replica to retain an additional old certificate for the request at $currentC.ts - 2$ so that it can vote for a candidate at $currentC.ts - 1$. If the replica has any votes to send, it sends an $\langle$OK, $V,\ h_c\rangle_{\sigma_r}$ message to all replicas, where V is the collection of votes and h_c is a hash of the CHECK message.

4. If the candidates include a certificate for $currentC.ts + 1$ which is valid for the replica, it is removed from the set of candidates. This certificate was included in the candidate set during the voting step, but now removed. It is important to vote for an invalid certificate if the replica contributed a grant for the operation, since there may be another replica that has seen a valid certificate for the operation, and is awaiting votes. Replicas do not *collect* votes for an invalid certificate however, since this may result in a quorum of votes with no valid certificate.

5. If a replica has a nonempty set of candidates, it waits for $f + 1$ OK *matching* messages with votes for the timestamp of its latest candidate certificate, or a later one. Two OK messages match at a timestamp t if each contains an identical vote for that timestamp. When the replica has the votes, it sends an $\langle$ACCEPT, $O,\ t,\ h_c\rangle_{\sigma_r}$ to the primary, where O contains the collection of supporting OK messages for timestamp t. If the set of candidates is empty, the replica doesn't wait for votes; instead it immediately sends an ACCEPT message, but in this case O is empty. Note that it is always safe for a replica to wait for $f + 1$ matching votes, since each candidate certificate remaining in the set appears valid to the replica, and contains grants from at least $f + 1$ honest replicas.

6. The primary waits for a quorum of valid ACCEPT messages, all containing an h_c that matches the hash of the CHECK message it sent earlier. It then calls BFT to run a CHECKEDRESOLVE operation, similar to a RESOLVE, with *startQ* and the collection of ACCEPT messages as arguments. Again, the entries in *startQ* are in the order the primary originally processed them.

8.2.3 Step 3

Replicas process the upcall from BFT as described in Chapter 4, except that in the case of a CHECKEDRESOLVE operation they use the ACCEPT messages to determine the latest certificate C. C is determined as the certificate in $startQ_{sub}$ that contains the largest timestamp mentioned in one of the ACCEPT messages, assuming it contains a valid nonempty O.

If the replica is out of date with respect to C, it must first perform state transfer to obtain any missing requests. If the replica were processing a RESOLVE upcall, it would fetch state up to $C.ts - 1$ as discussed in Section 4. If it is processing a CHECKEDRESOLVE upcall however, it uses the ACCEPT messages, included as an argument, to decide what to do. In this case, the ACCEPT message for C identifies the operation that should run at $C.ts - 1$. If this operation is in *startQ.ops*, the replica requests state transfer for requests up to $C.ts - 2$; otherwise it requests state transfer for requests up to $C.ts - 1$.

After bringing itself up to date, the replica forms the set L of additional operations to be executed, but it adds an operation to L only if the operation appears at least $f + 1$ times in *startQ.ops*. This ensures the operation was correctly received by at least one honest replica, and not fabricated by a malicious replica.

8.3 Correctness

In this section we discuss the safety and liveness of the symmetric key contention resolution protocol. As in Chapter 7 we assume that if a replica sends a message repeatedly, it will eventually be delivered; we also assume there are at most f faulty replicas.

We begin by stating four lemmas regarding Step 1 of the protocol.

Lemma 8.3.1 *$startQ_{sub}$ contains no conflicting proposals.*

Proof Obvious, by construction.

Lemma 8.3.2 *The set $startQ_{rejects} = startQ - startQ_{sub}$ contains at least k requests from faulty replicas.*

Proof START messages are "added" to $startQ_{rejects}$ in pairs, and this happens only when the two messages conflict. Honest replicas will never produce conflicting certificates, and therefore at least one of the two replicas that sent the conflicting pair is a liar.

Lemma 8.3.3 *The set $startQ_{rejects}$ contains at most k (defined in Section 8.2.1) messages from honest replicas.*

Proof Follows directly from Lemma 8.3.2.

Lemma 8.3.4 *When Step 1 terminates, $startQ_{sub}$ contains at least $f + 1$ messages from honest replicas.*

Proof When Step 1 terminates, $|startQ_{sub}| = 2f + 1 - k = (f + 1) + (f - k)$. From Lemma 8.3.2 we know that the primary has processed at least k messages from faulty replicas. If $k = f$, i.e., all messages from liars are in $startQ_{rejects}$, then all messages in $startQ_{sub}$ are from honest replicas and there are exactly $f + 1$ of them. Otherwise, $startQ_{sub}$ might contain up to $f - k$ messages from liars, but it still contains an additional $f + 1$ messages from honest nodes. Therefore $startQ_{sub}$ contains at least $f + 1$ messages from non-faulty replicas.

8.3.1 Safety

Our correctness condition has three parts:

- *Commitment.* The latest certificate, C, selected by the protocol must satisfy $C.ts \geq t$, where t is the timestamp of the most recently committed request. This way we ensure that all committed requests retain their place in the order.
- *Validity.* All requests executed at timestamps greater than t must be valid requests, i.e., ones requested by a client.
- *Consistency.* Each operation in $startQ$ that is assigned a timestamp greater than t, is assigned the same timestamp at all replicas.

We address these components separately.

Commitment.

The commitment condition is satisfied because the certificate corresponding to the most recently committed request will be present in $startQ_{sub}$. The selection process in Step 2 will select a timestamp for the next operation based upon this most recently committed request.

In Step 1, as many as half of the messages in $startQ_{rejects}$ might be from honest replicas. A concern may be that as a result we might not have a message in $startQ_{sub}$ from an honest replica that knows about the most recently committed request. However, by Lemma 8.3.4 we know that when Step 1 terminates, $startQ_{sub}$ contains at least $f + 1$ messages from honest replicas ($2f + 1 - k$ messages, for k faulty replicas). It therefore has a non-empty intersection with the set of at least $f + 1$ honest replicas that processed the most recently committed request. Since any replica in the intersection is honest, it will propose a certificate at least as recent as the most recently committed request.

The honest replicas that have knowledge of the most recently committed request will succeed in collecting votes for this or a newer request in Step 2. The primary will receive an ACCEPT message from at least one of these replicas, since it must wait for

$2f+1$ ACCEPTs, and hence include a response from at least one of the $f+1$ replicas with knowledge of the committed certificate.

Validity.

Validity was straightforward when signatures were used in place of authenticators, since the signatures in client requests determined the validity of a client request. When using authenticators however, the voting protocol is used to ensure request validity.

The request in the latest certificate C is guaranteed to be valid, since C was either proposed by at least $f+1$ replicas in their START messages, or it was built from the grants in the START messages, or it was selected based on an ACCEPT message containing $f+1$ valid votes. In all three cases at least one honest replica must have vouched for C, which is sufficient to guarantee that the request in C is valid.

All requests that are assigned timestamps greater than $C.ts$ must be valid since they must appear at least $f+1$ times in *ops*. Hence we only need to examine validity for requests assigned timestamps less than $C.ts$. Validity is also clear for requests with timestamps less than or equal to t, since these have already committed. So we are concerned only with requests that are assigned timestamps greater than t but less than $C.ts$.

If C was selected without voting on ACCEPT messages, then either $C.ts = t$ or $C.ts = t+1$; in this case there are no requests with timestamps greater than t but less than $C.ts$.

If C is selected by considering ACCEPT messages however, it is possible that $C.ts > t+1$. In this case C must be invalid and has been proposed by a faulty replica. Here a faulty replica has succeeded in getting at least one non-faulty replica to vote for C.

This implies two things:

1. The request in C must be valid, since no non-faulty replica will vote for it otherwise

2. $C.ts \leq t+2$, since *currentC* at a non-faulty replica cannot be more than

one timestamp ahead of the most recently committed request, and a non-faulty replica will not vote for a certificate with timestamp greater than $currentC.ts + 1$.

When $C.ts = t+2$, the request chosen to execute at timestamp $t+1$ is the request with its hash included in the votes for C. Since at least one of these votes comes from an honest replica, we are guaranteed that the request is valid.

8.3.2 Consistency

The ordering of each operation following $C.ts$ is defined by the set L, formed in Step 3 of contention resolution. This set is computed deterministically by each replica, with existence of an operation in L predicated on at least $f+1$ copies of an operation in $startQ.ops$. Given that each replica has $currentC.ts = C.ts$, and the ordering of operations in L is deterministic, each operation in L will be assigned the same timestamp across all replicas.

8.3.3 Liveness

There are two area where one may have concerns about the liveness of the contention resolution protocol—the termination of Steps 1 and 2.

Step 1

The termination of Step 1 is guaranteed since whenever the primary waits for another START message there is always at least one more non-faulty replica to send that message. The termination condition for Step 1 is $|startQ_{sub}| + k = 2f + 1$. By Lemma 8.3.3 we know that the maximum number of honest nodes with entries in $startQ_{rejects}$ is no greater that k. Therefore the number of honest replicas heard from so far is no more than $|startQ_{sub}| + k$, which, if we haven't yet reached termination, is less than $2f + 1$. It is thus safe for the primary to wait for another message, since there is at least one honest replica it hasn't heard from yet.

Step 2

All honest nodes will send valid ACCEPT messages to the primary, and thus Step 2 will terminate when the primary receives these $2f+1$ messages. If an honest replica has no candidate certificate for which it is waiting for votes, it sends an ACCEPT message immediately. Replicas that wait for votes do so only for certificates that appear valid to them. This implies that the certificate contains grants from at least $f+1$ honest replicas and those replicas will vote for either that certificate or a later one. Thus any replica that is waiting for votes will receive them, and be able to send an ACCEPT message to the primary.

Note that it is important for replicas to vote for all (possibly several) valid candidate timestamps. This addresses a situation that arises when some honest replicas have processed a WRITE-2 message for a later request, and hence hold a *currentC* certificate that is more recent than that known by the remaining honest replicas. If replicas only voted for their most recent *currentC*, it may not be possible to obtain a full quorum of matching votes.

It is also important that replicas accept matching votes for a certificate later than their candidates. This handles the case where an honest replica is behind the latest system state, and waiting for votes for candidate certificates that are older than the most recently committed operation. The replica will not receive a quorum of votes for its candidates, but may accept matching votes for the newer certificate.

We note that a given round of contention resolution may not succeed in executing any operations, since there may not be any operation that is known by the $f+1$ valid replicas participating in the voting phase. Any legitimate client that requests contention resolution will send a RESOLVE to all replicas however, containing the write operation that triggered contention. Once receiving this message, a replica will add the operation to *ops*, and vote for the operation in any subsequent contention resolution round. Thus progress is guaranteed once each replica has processed the RESOLVE request from a particular client.

8.4 State Transfer with Authenticators

The state transfer protocol, described in Chapter 6, requires modifications to function correctly when using authenticators. We can no longer guarantee that *backupC* and *currentC* certificates from an honest replica will appear valid to other replicas, and must adjust the protocol to accommodate this. Snapshots and log segments contain no certificates, and hence function identically under the authenticator scheme.

State transfer using authenticators is more conservative than in the signature protocol, in that it may only advance a slow replica's state to the lowest *backupC*, rather than *currentC*. This does not pose a liveness problem, however, since the slow replica will always advance at least up to the *newC* certificate that triggered state transfer.

If the *backupC* and *currentC* certificates in the shortest log segment (at least as long as $newC.ts - 1$) appear valid to the replica, then they may be processed as normal, according to the standard state transfer protocol. If the authenticators in these certificates do not appear valid however, additional processing is required.

When replacing signatures with authenticators, clients are no longer able to verify the integrity of a certificate. Moreover, the certificates themselves do not provide proof that an operation is valid during contention resolution; rather the proof is provided by votes during the voting phase. Hence it is sufficient for a replica to store *backupC* and *currentC* certificates that don't appear to contain a quorum of valid MACs, so long as the replica is able to guarantee that the certificates represent valid states.

The replica waits for $f + 1$ matching snapshots and log segments up to at least $new.ts - 1$. As in Chapter 6, the *currentC* response from a replica may be used in place of the final log entry, if the replica has $currentC.ts = newC.ts - 1$. The matching log segments may extend up to $newC.ts - 1$, or may continue beyond $newC.ts - 1$; these two cases handled separately below.

- If the log segment does not extend beyond $newC.ts - 1$, then *backupC* at the slow replica is replaced by the certificate corresponding to the end of the shortest log segment. The replica may then execute the operation corresponding to *newC*, which will appear in the *ops* response from a valid replica. We are guaranteed that the operation corresponding to *newC* will appear in *ops* from at least one replica, since *newC* contains grants from at least $f + 1$ honest replicas, each of which will store the operation in *ops*.

- If the certificate corresponding to the end of the shortest log segment has $ts \geq newC.ts$, then the system has progressed beyond *newC.ts*, and is not stalled waiting for a response to *newC*. The certificate at the end of the log segment must be valid and have committed, since it matches f other log entries, and hence it may be used to replace *currentC* at the slow replica.

  We are unable to replace *backupC* at the slow replica, however, as we may not receive any certificate for this previous timestamp. Instead, *backupC* is set to *null*. We discuss modifications to the protocol to accommodate a *null* backupC, in the following section.

8.4.1 ***null*** *backupC*s

As noted previously, a replica may have a *null backupC* as a result of state transfer. *backupC* is ordinarily used to replace *currentC* when it is rolled back during contention resolution. This is not a concern here since the *currentC* obtained in the second scenario of state transfer is retrieved from the *backupC* in $f + 1$ matching log segments, and is guaranteed to have committed, hence will not be rolled back.

A replica is free to respond to write requests while holding a *null backupC*, and will replace this with *currentC* when the next operation is executed.

backupC is used during contention resolution in Section 8.2.2, in Step 2 of the *startQ* validation protocol. While a replica with a *null backupC* will be unable to perform the check as described, it is able to retrieve the corresponding *backupC.ts* and *backupC.h* information from its log to perform the equivalent check.

The final situation where *backupC* is usually required is in the response to a state transfer request. The replica may have to respond to a state transfer request to facilitate liveness; this is necessary where the replica sends a grant to a client for $currentC.ts + 1$, the client performs a WRITE-2 at a slow replica, and a state transfer response is required to bring it up to date.

The replica sends its STATE response as before, but with a *null* $backupC$. The log in the response will cover timestamps up to and including $currentC.ts$. If the system is stalled and the STATE responses extend to $newC.ts - 1$, as in the first scenario above, then the replica's $currentC$ will contribute towards $backupC$ at the slow replica. The slow replica will assign $newC$ to $currentC$, and retrieve the corresponding operation from ops. If the scenario is instead the second case, where log segments extend beyond $newC$, then the replica must have executed operations above $currentC.ts$, in which case $backupC$ will no longer be *null*. Hence in both cases state transfer is possible regardless of the *null* $backupC$.

Chapter 9

Performance Evaluation

We examine here the performance characteristics of the HQ protocol, with respect to BFT and Q/U, the two canonical agreement-based and quorum-based replication protocols respectively.

Our focus is on write-only workloads. All three protocols offer efficient one-phase read optimizations, and we expect similar performance for reads. Write operations require running of the replication protocols, and place far higher load on the systems themselves.

9.1 BFT Improvements

We first discuss the performance improvements made to BFT in the process of evaluating the performance of HQ. These changes improve the scalability of the BFT protocol, and are implemented to provide a strong benchmark to compare our performance against.

The original implementation of BFT, while very comprehensive, was optimized for high performance in broadcast environments and with low value of f, e.g. $f = 2$. While BFT performance has been criticized, particularly with regard to throughput at high f [1], much of the limitations observed have been an artifact of the implementation itself and not the BFT protocol. In this work we place particular emphasis on scalable performance with increasing f, and hence make a number of modifications to the BFT implementation.

The first modification is the use of MACs rather than authenticators in agreement protocol messages. The original BFT implementation was designed to be deployed on a single network with IP multicast, broadcasting the same message to each replica, hence requiring authenticators so that a single message can be verified by all replicas. If we deploy BFT in a more common distributed environment, however, with unicast communication between replicas, these authenticators are not required, and add linearly scaling overhead to each message. Instead we replace authenticators with a single MAC per message, greatly reducing message size for large replica sets.

As with HQ, we also extend BFT to support preferred quorums under failure-free operation. The benefits of preferred quorums are more significant in BFT than HQ or Q/U, as preferred quorums yield an approximately one-third reduction in the number of replicas involved in the quadratic communication phases of the protocol.

Along with the aforementioned protocol modifications, we redeployed the BFT code base with TCP communication, rather than UDP and manual flow control in the original implementation. This avoids costly message loss due to congestion at high f and maximum throughput. TCP is a more carefully engineered protocol than the original flow control implementation, and offers more predictable performance.

The performance benefits of these relatively small modifications is quite significant, as exhibited in Section 9.3.

9.2 Analysis

We begin our evaluation with a theoretical analysis of HQ, BFT and Q/U. This affords us a direct comparison with the Q/U protocol. We were unable to obtain the Q/U source code to examine its performance experimentally.

Focus here is on performance in the optimistic case of no write contention and no failures. For both HQ and Q/U we assume the use of preferred quorums and MACs/authenticators rather than signatures. We also assume the optimistic case of one-phase writes under Q/U, rather than the two phases required when client knowledge of system state is stale. These results hence give an upper bound for Q/U performance, and a lower bound for its overhead. Three variants of BFT are examined: the original *BFT* algorithm, using authenticators and without preferred quorums; *BFT-MACs*, using MACs rather than authenticators, but no preferred quorums; and *BFT-opt*, using both MACs and preferred quorums. We assume that all protocol communication is done via point-to-point message exchange, rather than IP multicast.

Figures 9-1 and 9-2 show the communication patterns for BFT and HQ respectively; the communication pattern for Q/U is similar to the first phase of HQ, with a larger number of replicas ($5f + 1$). We assume here that write latency is dominated by the number of message delays required to process a request, as will be the case for write operations requiring relatively little application level processing. We thus observe that the latency of HQ is lower than that of BFT, and the latency for Q/U half of that for HQ. One point to note is that BFT can be optimized so that replicas reply to a client request following the *prepare* phase, eliminating *commit*-phase latency in the absence of failures; with this optimization, BFT can achieve the same latency as HQ. To amortize its quadratic message costs however, BFT employs batching, committing a group of operations as a single unit, as examined in Section 9.3.4. This can lead to additional latency over a quorum-based scheme with lower message overhead.

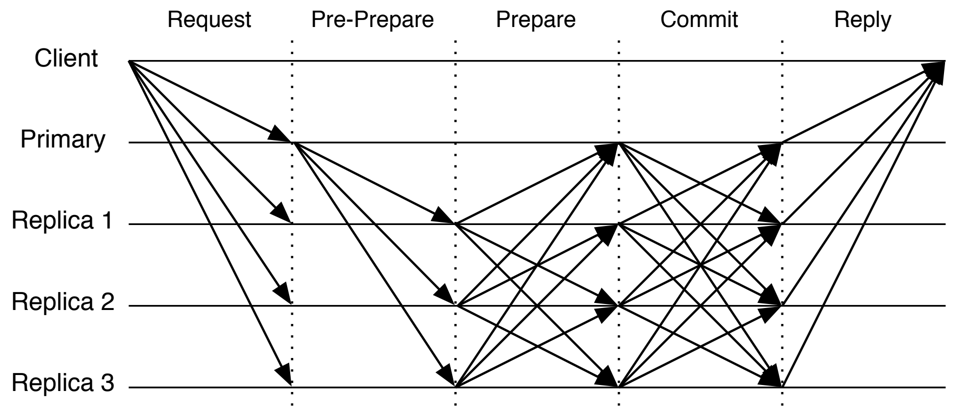

Figure 9-1: Agreement-based protocol communication pattern (BFT).

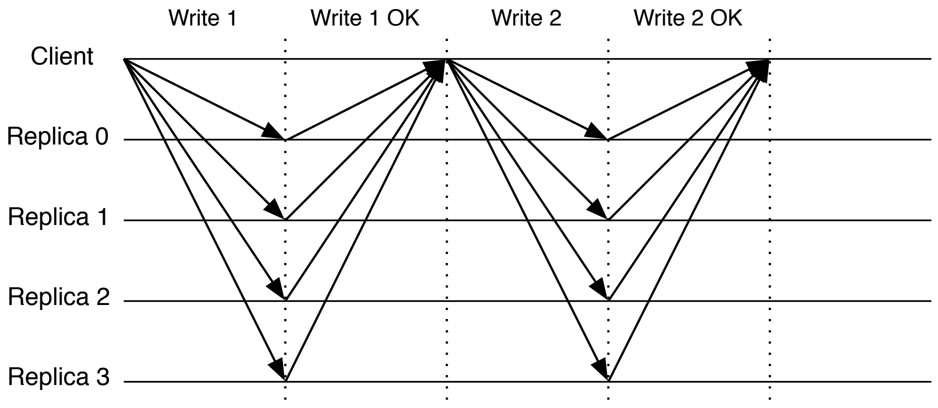

Figure 9-2: Quorum-based protocol communication pattern (HQ).

Figures 9-3 and 9-4 show the total number of message exchanges required to order a write request in the three systems; the figure shows both the load per server and the load per client. BFT-MACs is not shown as it has the same message load as BFT-opt. Consider first Figure 9-3, which shows the load at each server. In both HQ and Q/U, servers process a constant number of messages to carry out a write request: 4 messages in HQ and 2 in Q/U. In BFT, however, the number of messages is linear in f: For each write operation that is ordered by BFT, each replica must process $12f + 2$ messages. This is reduced to $8f + 2$ messages in BFT-opt through our use of preferred quorums.

Message load at each client is shown in Figure 9-4. We see that BFT-opt has the lowest cost, only requiring a client to send a request to all replicas, and receive a quorum of replies. Q/U also requires only one message exchange, but has larger quorums, for a total of $9f + 2$ messages ($5f + 1$ requests, $4f + 1$ replies). HQ has two message exchanges but has quorums of size $2f + 1$; therefore the number of messages processed at the client is $9f + 4$ ($3f + 1$ phase 1 requests, $2f + 1$ phase 1 replies, phase 2 request and phase 2 replies), similar to Q/U.

A different picture is portrayed in Figures 9-5 and 9-6, which take into consideration the actual sizes of the messages involved in protocol communication. We compute these sizes using 20-byte SHA-1 digests [33] and HMAC authentication codes [32], 44

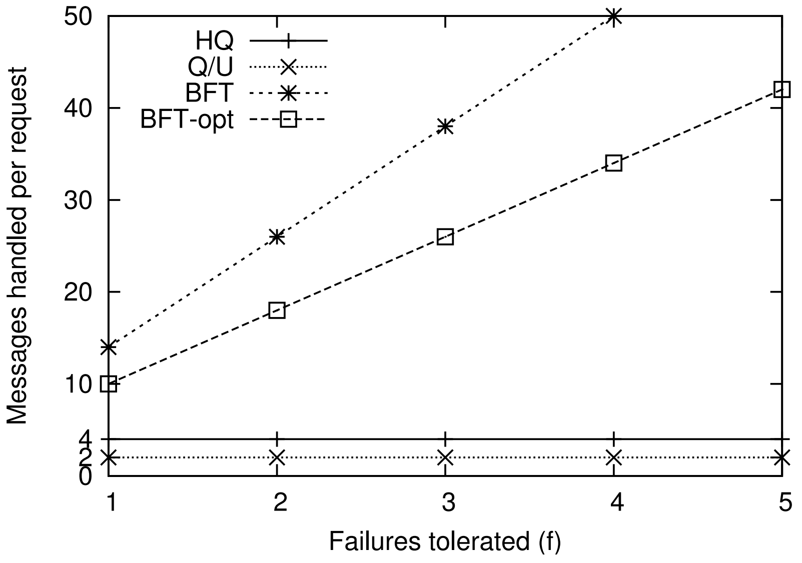

Figure 9-3: Total messages sent/received at each server per write operation in BFT, Q/U, and HQ

byte TCP/IP overhead, and a nominal request payload of 256 bytes. Analyze of Q/U is done on the fully optimized version, using *compact timestamps* and *replica histories* pruned to the minimum number of two *candidates* [1].

The results for BFT in Figure 9-5 show that our optimizations (MACs and preferred quorums) have a major impact on the byte count at replicas. The use of MACs rather than authenticators causes the number of bytes to grow only linearly with f as opposed to quadratically as in BFT, as shown by the BFT-MACs line; an additional linear reduction in traffic occurs through the use of preferred quorums, as shown by BFT-opt line. We note that BFT was originally designed for a broadcast environment, and hence the linearly scaling message sizes resulted only in a linearly scaling communication load per replica, as each replica sends a constant number of messages with respect to f. When deployed on a unicast communication fabric however, as is common in modern incarnations, each replica sends a linearly scaling number of messages, leading to quadratic communication load.

Figure 9-5 also shows results for HQ and Q/U. The responses the WRITE-1 requests in HQ contain an authenticator, with WRITE-2 requests containing a certifi-

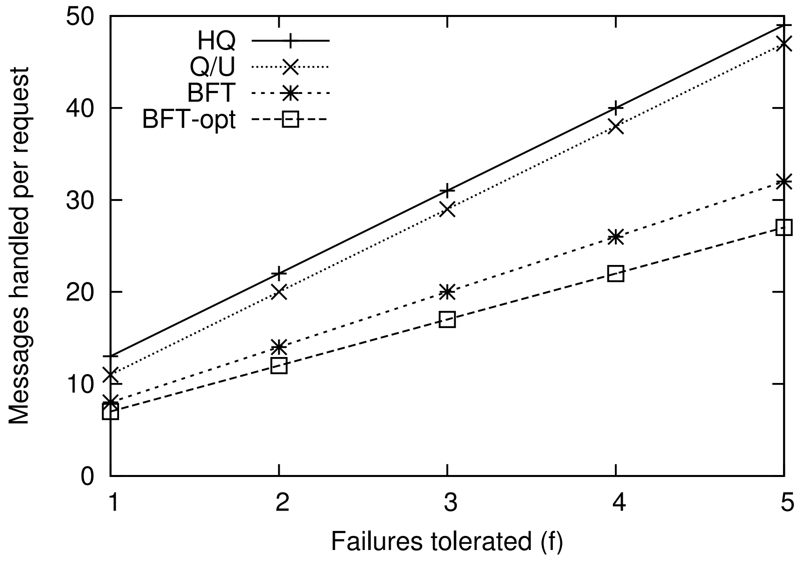

Figure 9-4: Total messages sent/received at each client per write operation in BFT, Q/U, and HQ

cate, both growing quadratically with f. Q/U is similar: The response to a write returns what is effectively a grant (*replica history*), with these combined to form a certificate (*object history set*), which is sent in the next write request. The grants in Q/U are considerably larger than those in HQ however, and also contain larger authenticators (size $4f + 1$ instead of $2f + 1$), resulting in more bytes per request in Q/U than HQ. While HQ and Q/U are both affected by quadratically-sized certificates, this becomes a problem more slowly in HQ. At a given value of $f = x$ in Q/U, each certificate contains the same number of grants as in HQ at $f = 2x$.

Bandwidth required at the client in bytes/request is illustrated in Figure 9-6. Client load for BFT is low, since the client simply sends a request to all replicas and receives a quorum of responses. The load for Q/U is the highest, owing to the quadratically growing certificates, larger grants, and communication with approximately twice as many replicas.

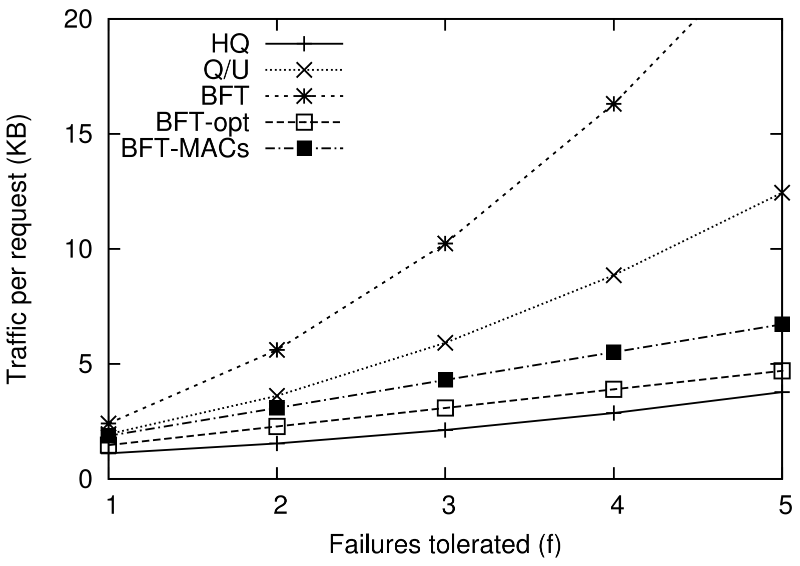

Figure 9-5: Total traffic sent/received at each server per write operation in BFT, Q/U, and HQ

9.3 Experimental Results

This section provides performance results for HQ and BFT in the case of no failures.

9.3.1 Experimental Setup

We focus on write performance in a simple counter service, supporting increment and fetch operations as in Q/U [1]. This results in negligible operation cost, allowing us to focus on protocol overhead itself. The system supports multiple counter objects. Each client request operates on a single object and the client waits for a write to return before executing any subsequent request. We vary the level of write contention by the use of two types of objects—a private object for each client, and a single shared object for all. In the non-contention experiments different clients operate on their own independent objects, while in the contention experiments a certain percentage operate on the shared object, to obtain the desired level of write contention.

To allow meaningful comparisons of HQ and BFT, we produced new implementations of both, derived from a common C++ code base. Communication is imple-

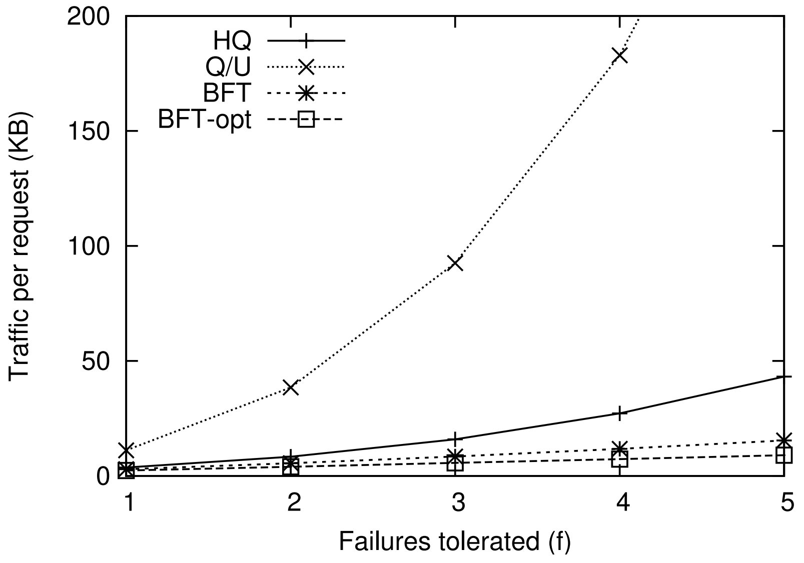

Figure 9-6: Total traffic sent/received at each client per write operation in BFT, Q/U, and HQ

mented over TCP/IP sockets, and we use SHA-1 digests for HMAC message authentication. HQ uses preferred quorums; BFT-MACs and BFT-opt use MACs instead of authenticators, with the latter running preferred quorums. Client operations in the counter service consist of a 10 byte *op* payload, with no disk accesses required in executing operations.

The performance experiments run on Emulab [46], utilizing 66 *pc3000* machines. These contain 3.0 GHz 64-bit Xeon processors with 2GBs of RAM, each equipped with gigabit NICs. The emulated topology consists of a 100Mbps switched LAN with near-zero latency, hosted on a gigabit backplane with a Cisco 6509 high-speed switch. Network bandwidth was not found to be a limiting factor in any of the experiments. All machines run Fedora Core 4, with Linux kernel 2.6.12.

Sixteen machines of the Emulab machine are used to host a single replica each, providing support for up to $f = 5$, with each of the remaining 50 machines hosting two clients. The number of logical clients is varied between 20 and 100 in each experiment, to obtain maximum possible throughput. This large number of clients is needed to fully load the system since clients are restricted to only one operation at a time.

Each experiment runs for 100,000 client operations of burn-in time to allow performance to stabilize, before recording data for the following 100,000 operations. Five repeat runs were recorded for each data-point, with the variance too small to be visible in the following plots. The main performance metric observed in these experiments is maximum system throughput. Message count and bandwidth utilization were observed and found to closely match the analysis presented in the previous section.

9.3.2 Non-contention Throughput

Our first experiments are performed in the absence of write contention, observing maximum throughput along with scalability with increasing f. We keep BFT batch size at 1; batch size is explored in Section 9.3.4. System throughput is found to be CPU bound in all experiments, owing to message processing expense and cryptographic operations, along with kernel message handling overhead. The network itself poses no limiting effects—bandwidth limits are not reached, and the number of clients is kept sufficiently high to ensure that request load is not limited by RTT or the delay in completing requests.

Figure 9-7 shows higher throughput in HQ than the BFT variants. This is a result of the lower message count and fewer communication phases required by the HQ protocol. The figure also shows significant benefits for the two BFT optimizations: the reduction in message size achieved by BFT-MACs, and the reduced communication and cryptographic processing costs in BFT-opt.

Throughput in HQ drops by 50% as f grows from 1 to 5, a consequence of the overhead of computing larger authenticators in grants, along with receiving and validating larger certificates. The BFT variants show slightly worse scalability, due to the quadratic number of protocol messages. We note from the previous analysis that both the HQ and optimized BFT protocols scale similarly in the volume (in bytes) of protocol communication. This volume is spread over a constant number of increasingly-sized messages in HQ however, with an increasing number of constant-sized messages in the optimized variants of BFT. This greater number of messages to process in BFT leads to higher CPU overhead than in HQ, owing to the higher kernel overhead in processing each message.

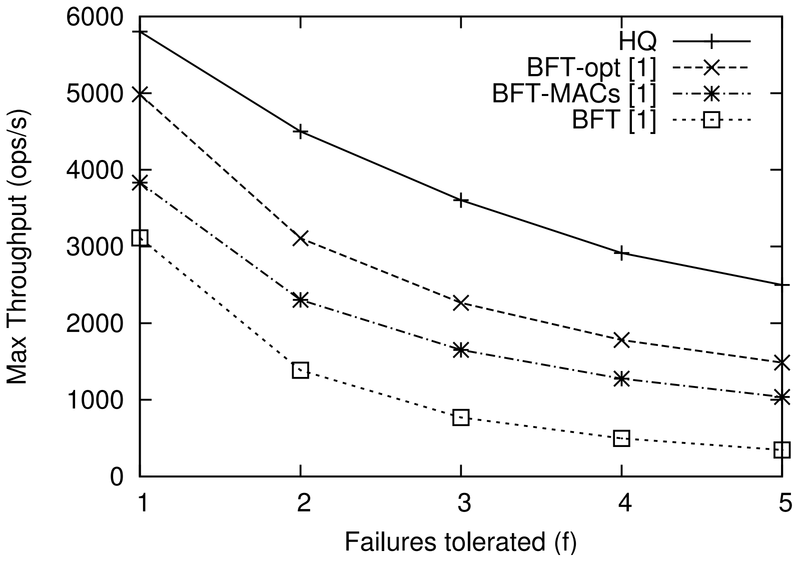

Figure 9-7: Maximum non-batched write throughput under varying f.

Based on the analysis in Section 9.2, we expect Q/U to provide somewhat less than twice the throughput of HQ at $f = 1$. It requires half the server message exchanges but more processing per message, owing to larger messages and more MAC computations. We also expect Q/U to scale less well than HQ, since its message exchanges and processing grow more quickly with f than HQ's.

9.3.3 Resilience Under Contention

Of particular interest is performance in the presence of write contention, as this is an area handled poorly by previous quorum protocols [1]. When multiple write requests are issued concurrently it becomes far less likely that a single client will receive a matching quorum of grants. Agreement-based protocols such as BFT are unaffected by contention however, since ordering is computed by a single entity—the primary. HQ employs the use of BFT during contention to capitalize on these performance benefits.

Figure 9-8 shows maximum throughput for HQ with $f = 1, ..., 5$ in the presence of write contention. *Contention factor* as presented in the figure is the fraction of writes executed on a single shared object. At contention factor 1, all writes execute on the same object, requiring contention resolution or writeback operation in almost all cases. The figure shows that HQ performance degrades gracefully as contention increases. Throughput reduction flattens significantly at high rates of write contention because multiple contending operations are included in the same *startQ*; hence they are ordered with a single round of BFT, achieving a degree of write *batching*. At $f = 2$ with contention factor 0.1, an average of 3 operations are ordered per round of contention resolution, while at contention factor 1 this increases to 16 operations per round.

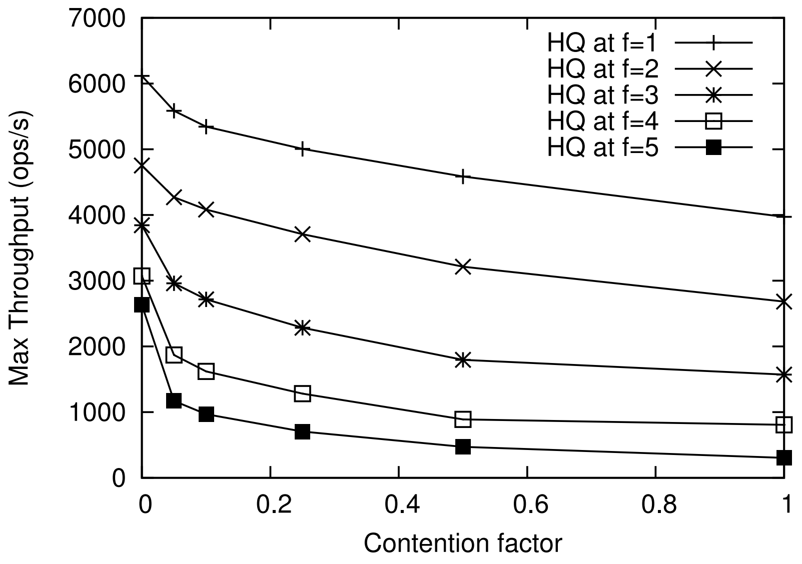

Figure 9-8: HQ throughput under increasing write contention.

BFT performance under contention is as presented in Figure 9-7. Performance crosses over between HQ and BFT at around 25% contention, beyond which BFT outperforms HQ.

9.3.4 BFT Batching

In the previous section we saw that HQ throughput benefits from the inclusion of multiple operations in the same round of contention resolution. This technique is possible on a larger scale in BFT through the use of *batching* [7]. Under batched operation, the BFT primary buffers a number of client write requests, running the agreement protocol only once for the whole group. This is a technique possible in all primary-driven agreement protocols, but not possible under quorum schemes such as HQ, owing to the lack of a centralized coordinator to order operations within a batch. Batching has the potential to greatly reduce internal protocol communication, increasing throughput, albeit with additional client delay. The additional delay is low when system load is high however, since the high rate of client requests allows the primary to send batches at high frequency. Since batching is used primarily to reduce protocol overhead at high load, the overhead in client delay is typically low.

Figure 9-9 shows the effect of batching on BFT performance, for our optimized version of BFT. The number in brackets in the legend corresponds to the number of requests batched in each round of the protocol, for 1, 2, 5, and 10 messages. We see a significant improvement in throughput with increasing batch size, surpassing HQ above a batch size of 2. A batch size of 10 incurred less than 5ms additional delay, with the system loaded by 100 continuously active clients.

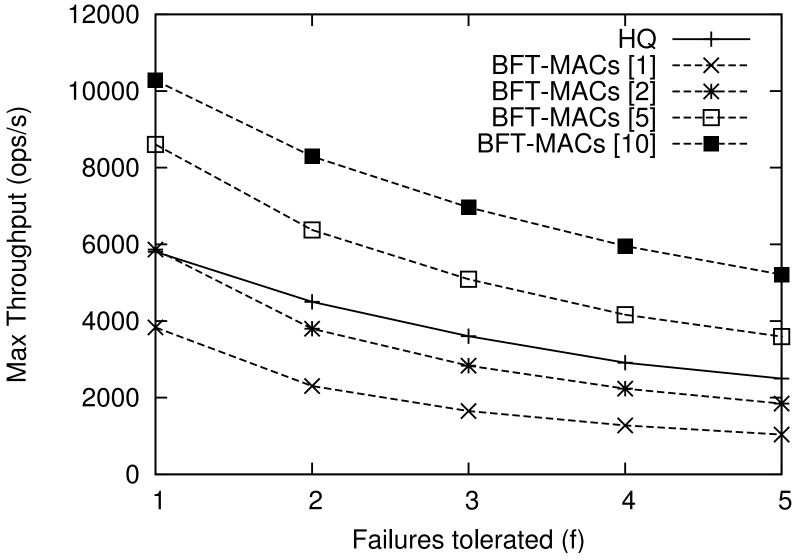

Figure 9-9: Effect of BFT batching on maximum write throughput.

Chapter 10

Related Work

There is a vast body of research in the areas of fault tolerance and state machine replication. We present a brief overview of replication protocols for tolerating fail-stop faults, and protocols that provide Byzantine-fault-tolerance with weaker semantics than HQ, such as assuming synchrony or offering a restricted operations interface. Our main focus, however, is on Byzantine-fault-tolerant state machine replication protocols that provide support for general operations in an asynchronous environment, such as the BFT [7] and Q/U [1] protocols.

10.1 Fail-stop Fault Tolerance

Initial proposals for fault tolerant replicated systems assumed a *fail-stop* or *benign* fault model [11, 43, 34, 16]. In such a model, replicas may fail by stopping or by disconnecting from the network, but must otherwise behave correctly.

Notable of these protocols are the Oki and Liskov Viewstamped Replication protocol [34], later deployed in the Harp system [21], and Lamport's Paxos algorithm [16]. While Viewstamped Replication implements state machine replication [44] (for ordering multiple client operations) and Paxos is presented purely as an agreement algorithm (reaching agreement on only a single operation), they are in essence very similar. Both protocols utilize a primary-backup mechanism [2] to facilitate agreement, where a primary replica assigns a sequence number to client requests. A view-change protocol is used to replace a faulty primary, utilizing quorums [11] to ensure that existing ordering information persists with the new primary.

10.2 Byzantine Fault Tolerance

Byzantine-fault-tolerant replication protocols make no assumptions about the behavior of replicas, and assume they may act arbitrarily or maliciously [35, 17]. Initial proposals [35, 17, 42, 10, 13] assumed that communication is synchronous, which is not the case in an environment such as the Internet, but more recent protocols support an asynchronous communication model.

Protocols for Byzantine-fault-tolerant services are generally divided into two approaches—a primary-driven agreement (state machine replication) approach, or a client-controlled quorum approach.

10.2.1 Byzantine Agreement

Agreement based protocols follow a primary-backup model for replication, where a primary replica assigns a sequence number to all operations. Pure agreement or *consensus* protocols provide agreement on a single operation [3, 5, 4]. As such they form the foundation for protocols such as BFT, but do not support state machine replication for multiple sequential operations.

State machine replication protocols are designed to support agreement on a continual series of client operations. The Rampart [36] and SecureRing [13] protocols implement state machine replication for synchronous networks. These protocols rely on group communication to reach consensus, and use failure detectors to remove faulty replicas from the consensus group. Accurate failure detection is not possible in asynchronous networks, however [23].

The Castro and Liskov BFT protocol [7] supports state machine replication in asynchronous networks, and is used by the HQ protocol to provide support for concurrent write requests. Like previous state machine replication protocols [36, 13], BFT requires the theoretical minimum of $3f + 1$ replicas to support f faults. The BFT agreement protocol proceeds in three phases of inter-replica communication, imposing communication overhead that scales quadratically in f. A significant benefit of BFT, as distinct from previous protocols, is that it supports the use of symmetric key cryptography, rather than relying on expensive computationally public key signatures. The Q/U and HQ protocols both support the use of symmetric key cryptography.

Primary-driven state machine replication protocols provide a single point of serialization for each client operation, and are hence able to offer consistent performance when writes are issued concurrently. They also support the use of batching, exploited for significant performance gain in BFT, to amortize the cost of running the agreement protocol over a number of operations.

10.2.2 Byzantine Quorums

Byzantine quorum systems were introduced by Malkhi and Reiter [26]. The basic algorithm behind these systems involves clients leading the protocol through a sequence of phases; in each phase a client reads or writes data to a quorum, and the quorum intersection property [11] guarantees that reads will always return the outcome of the most recently completed write (or a concurrent write).

The initial Byzantine quorum constructions were designed to handle only read and blind write operations. These are less powerful than state machine replication with general operations (provided by HQ), where the outcome of an operation can depend on the history of previously executed operations. Byzantine quorum protocols were later extended to support complex operations with benign clients in Phalanx [25]. The subsequent Fleet system [27] prevents malicious clients from diverging system state, requiring $4f + 1$ replicas. Fleet has been further extended to support state machine replication [8], with four round trips per operation and a minimum of $5f + 1$ replicas ($6f + 1$ if symmetric key cryptography is used).

Rosebud [40] presents a Byzantine-fault-tolerant storage architecture, with a single-phase quorum protocol at its core, providing support for reads and blind writes. It requires only $3f + 1$ replicas but does not support Byzantine clients. The BFT-BC protocol [22] extends this work to build a three-phase quorum protocol (two-phase in the optimal case) with $3f + 1$ replicas that handles arbitrary client behavior. This research provided much of the motivation for the development of HQ.

The recent Q/U protocol [1], by Abd-El-Malek et al., presents a novel "optimistic" quorum-replication protocol for Byzantine-fault-tolerance. Q/U is designed to provide levels of performance similar to agreement-based approaches (such as BFT) for small system sizes, and outperform these protocols when scaling to large values of f. Writes execute in two-phases in Q/U, or one phase when only a single client is writing to an object. In the first phase the replica obtains a summarized history of system state from the replicas, and in the second phase provides a matching quorum of state history entries to each replica, requesting execution of the write operation at the next sequence number in this state. State histories are maintained at clients to facilitate repair when the state at replicas diverges.

Q/U requires $5f + 1$ replicas, and quorums of size $4f + 1$; this is shown to be the minimum number of replicas required for two-phase agreement in a quorum-replication scheme. Performance for Q/U degrades significantly when there are concurrent writes, where replicas accept writes from different clients at the same sequence number. The repair process used to establish consistent state involves clients performing exponential backoff while attempting establish consistent state; this repair process requires clients to perform a barrier operation at a quorum of replicas, halting progress, and to propagate state across replicas. The repair process greatly reduces system throughput where multiple clients are attempting concurrent writes.

Q/U also demonstrates for the first time how to adapt a quorum protocol to implement state machine replication for multi-operation transactions with Byzantine clients. This is an important development, since the application space in quorum schemes is often partitioned into multiple objects, to minimize performance degradation during concurrent writes. Another feature pioneered by Q/U is the use of *preferred quorums*, to constrain communication to a single quorum of replicas under normal operation. This reduces the active replica set from $5f + 1$ to $4f + 1$ in Q/U, and from $3f + 1$ to $2f + 1$ in both BFT and HQ.

In work performed concurrently with that on Q/U, Martin and Alvisi [28] discuss the trade-off between number of rounds and number of replicas for reaching *agreement*, a building block that can be used to construct state machine replication. They prove that $5f+1$ replicas are needed to ensure agreement is reached in two communication steps, and present FaB Paxos, a replica-based algorithm that shows this lower bound is tight. They also present a parametrized version of FaB Paxos that requires $3f+2t+1$ replicas—it is safe despite up to f Byzantine failures, and provides two-step consensus with up to t Byzantine failures. The bound of $3f+2t+1$ replicas implies their system can execute with $5f+1$ replicas and provide two-step consensus with up to f failures ($t=f$), or with $3f+1$ replicas but no longer provide two-step operation if there are any failures ($t=0$).

10.2.3 Hybrid Quorum Replication

Our HQ protocol builds on the work of Q/U and FaB Paxos, but reduces the number of replicas from $5f+1$ to $3f+1$, and does not suffer significant performance degradation during write contention. These improvements are made possible through the use of *Hybrid Quorum* replication, a new technique where BFT [7] is used to resolve write contention. By supplementing a quorum system with an agreement-based state machine replication protocol, we are able to combine the benefits of quorum schemes (two-phase writes, scalability to large f), with those of agreement-based protocols (resilience to write contention, $3f+1$ replicas). To our knowledge we are the first to combine these approaches into a hybrid protocol.

The BFT module included in HQ is used to reach agreement on a set of concurrent write operations, rather than assign an ordering to individual operations as in traditional state machine replication. In this regard it is similar to Byzantine consensus protocols, such as FaB Paxos [28].

Kursawe's Optimistic Byzantine Agreement protocol [14], provides support for both optimistic and pessimistic agreement, but is less flexible than the HQ protocol. This protocol provides two-step agreement with $3f+1$ replicas during "well-behaved" executions, during which messages are received within delay bounds and there are no replica failures. If the system deviates from a well-behaved execution, the protocol permanently switches to a traditional agreement protocol. HQ uses BFT to resolve instances of write contention, but returns to the optimistic quorum protocol for subsequent operations.

10.3 Further Developments

The recent Li and Mazières BFT2F protocol [20] addresses the question of what guarantees can be provided when more than f replicas fail in a Byzantine-fault-tolerant replicated system. Existing protocols provide no guarantees above f faults, and bad replicas may deliver incorrect state to clients without detection in both BFT and HQ. The authors of BFT2F modify the BFT protocol to prevent the fabricating operations beyond f failures, and support *fork** consistency between f and $2f$ failures, a weaker form of fork consistency [30].

Yin, Martin et.al., [47] develop a technique for separating the agreement process in Byzantine-fault-tolerant state machine replication from the execution of the actual operations. This is an important optimization, and allows agreement to be conducted on a set of $3f + 1$ replicas, while reducing the number of the more expensive storage and execution replicas to only $2f + 1$. We believe the same techniques may be applied to the HQ protocol, to reduce the number of storage replicas in the system.

Chapter 11

Conclusions

We have presented HQ Replication, a novel protocol for Byzantine-fault-tolerant state machine replication. HQ employs a new *hybrid quorum* approach, combining the benefits of a client-controlled, quorum-based protocol, with agreement-based classical state machine replication. Through this combination, HQ is able to provide low-overhead execution in the absence of contending updates, while efficiently serializing concurrent requests under agreement where required.

HQ offers greater resilience against contention collapse than previous quorum approaches, while avoiding the excessive communication of traditional agreement protocols. Furthermore, HQ reduces the required number of replicas from $5f + 1$ in earlier proposals [1, 29] to the theoretical minimum of $3f + 1$ replicas. This is important both from a practical standpoint, and to minimize the probability of failure within a replica group.

Under normal operation, HQ employs an optimistic approach to ordering client requests, using BFT to resolve instances of contention when this optimism fails. The hybrid quorum approach can be used broadly; for example it could be used in Q/U to handle write contention, where BFT would only need to run at a predetermined subset of $3f + 1$ replicas.

As part of our analysis we presented a new implementation of the BFT protocol, developed to scale effectively with f. We found this implementation to perform well, and not exhibit the scalability problems highlighted in earlier work [1].

Our performance claims are supported with theoretical and experimental analysis, comparing HQ with two notable quorum-based and agreement-based protocols respectively: Q/U and BFT. We summarize the conclusions of our analysis, and the trade-offs between HQ, Q/U and BFT, in the following three suggestions for system choice:

- In the region we studied (up to $f = 5$), if contention is low, low request latency is the primary concern, and it is acceptable to use $5f + 1$ replicas, Q/U is the best choice. It offers the lowest request latency under optimal conditions, albeit with higher bandwidth overhead. If the overhead of $5f + 1$ replicas is too high, then HQ is most appropriate, since it outperforms BFT with low batch size, while providing two-phase write operations.

- If maximum system throughput is the primary objective, or contention or failures are high, BFT is the best choice. BFT handles high contention workloads without any performance degradation, and can outperform HQ and Q/U in terms of throughput through the use of request batching.

- Outside of the studied region ($f > 5$), we expect HQ will scale best with increasing f. Our results show that as f grows, HQ's throughput decreases more slowly than both Q/U and BFT. HQ outperforms Q/U because of the latter's larger messages and processing costs, and beats BFT where batching cannot compensate for the quadratic number of protocol messages.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM Press.

[2] Peter A. Alsberg and John 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, October 1976.

[3] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–240, 1985.

[4] 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.

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

[6] Miguel Castro. Practical byzantine fault tolerance. Ph.D. Thesis MIT/LCS/TR-817, MIT Laboratory for Computer Science, Cambridge, Massachusetts, January 2001.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[8] G. Chockler, D. Malkhi, and M. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proc. of the IEEE International Conference on Distributed Computing Systems*, 2001.

[9] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington, November 2006.

[10] J. Garay and Y. Moses. Fully polynomial byzantine agreement for n ¿ 3t processors in t+1 rounds. *SIAM Journal of Computing*, 27(1):247–290, February 1998.

[11] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the Seventh Symposium on Operating Systems Principles*, December 1979.

[12] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, 1987.

[13] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 3*, page 317, Washington, DC, USA, 1998. IEEE Computer Society.

[14] K. Kursawe. Optimistic byzantine agreement. In *Proceedings of the 21st SRDS*, 2002.

[15] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.

[16] L. Lamport. The Part-Time Parliament. Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[17] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[19] Leslie. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[20] Jinyuan Li and David Mazires. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[21] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 226–238, Pacific Grove, California, 1991.

[22] Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. Technical Report MIT-LCS-TR-994 and INESC-ID TR-10-2005, July 2005.

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

[24] 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.

[25] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[26] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[27] Dahlia Malkhi and Michael Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, April 2000.

[28] J.-P. Martin and L. Alvisi. Fast byzantine consensus. In *International Conference on Dependable Systems and Networks*, pages 402–411. IEEE, 2005.

[29] Jean-Philippe Martin. Fast byzantine consensus. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 402–411, Washington, DC, USA, 2005. IEEE Computer Society.

[30] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[31] National Institute of Standards and Technology. Digital signature standard. NIST FIPS PUB 86, U.S. Department of Commerce, 1994.

[32] National Institute of Standards and Technology. Fips 198: The keyed-hash message authentication code (hmac), March 2002.

[33] National Institute of Standards and Tecnology. Fips 180-2: Secure hash standard, August 2002.

[34] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing (PODC 1988)*, Toronto, Ontario, Canada, 1988.

[35] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[36] 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.

[37] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[38] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[39] M. J. B. Robshaw. Md2, md4, md5, sha and other hash functions. Tech. Rep. TR-101, RSA Labs, 1995.

[40] Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. MIT LCS TR/932, December 2003.

[41] 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.

[42] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[43] Fred B. Schneider. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, 4(2):125–148, 1982.

[44] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[45] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM Computer Communications Review*, 22(5):29–38, 1992.

[46] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[47] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.