Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-009 February 12, 2007

HQ Replication: Properties and Optimizations

James Cowling, Daniel Myers, Barbara Liskov,
Rodrigo Rodrigues, and Liuba Shrira

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

HQ Replication: Properties and Optimizations

James Cowling[1], Daniel Myers[1], Barbara Liskov[1], Rodrigo Rodrigues[2], and Liuba Shrira[3]
[1]*MIT CSAIL,* [2]*INESC-ID and Instituto Superior Técnico,* [3]*Brandeis University*
{*cowling, dsm, liskov, rodrigo, liuba*}*@csail.mit.edu*

Abstract

There are currently two approaches to providing Byzantine-fault-tolerant state machine replication: a replica-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 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.

We 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.

Preface

This technical report is an extended version of Cowling et al [4]. The new material can be found in the appendices, which present a discussion of the liveness of our system and the full protocol for contention resolution when we avoid the expense of digital signatures in our base protocol.

1 Introduction

Byzantine fault tolerance enhances the availability and reliability of replicated services in which faulty nodes may behave arbitrarily. In particular, state machine replication protocols [9, 19] that tolerate Byzantine faults allow for the replication of any deterministic service.

Initial proposals for Byzantine-fault-tolerant state machine replication [18, 2] relied on all-to-all communication among replicas to agree on the order in which to execute operations. This can pose a scalability problem as the number of faults tolerated by the system (and thus the number of replicas) increases.

In their recent paper describing the Q/U protocol [1], Abd-El-Malek et al. note this weakness of agreement approaches and show how to adapt Byzantine quorum protocols, which had previously been mostly limited to a restricted read/write interface [12], to implement Byzantine-fault-tolerant state machine replication. This is achieved through a client-directed process that requires one round of communication between the client and the replicas when there is no contention and no failures.

However, Q/U has two shortcomings 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 $3f+1$. 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. Second, Q/U performs poorly when there is contention among concurrent write operations: it resorts to exponential back-off to resolve contention, leading to greatly reduced throughput. Performance under write contention is of particular concern, given that such workloads are generated by many applications of interest (e.g. transactional systems).

This paper presents the Hybrid Quorum (HQ) replication protocol, a new quorum-based protocol for Byzan-

tine fault tolerant systems that overcomes these limitations. HQ requires only $3f + 1$ replicas and combines quorum and agreement-based state machine replication techniques to 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, it uses the BFT state machine replication algorithm [2] to efficiently order the contending operations. A further point is that, like Q/U and BFT, HQ handles Byzantine clients as well as servers.

The paper additionally presents a new implementation of BFT. The original implementation of BFT [2] was designed to work well at small f; our new implementation is designed to scale as f grows.

The paper presents analytic results for HQ, Q/U, and BFT, and performance results for HQ and BFT. Our results indicate that both HQ and the new implementation of BFT scale acceptably in the region studied (up to $f = 5$) and that our approach to resolving contention provides a gradual degradation in performance as contention rises.

The paper is organized as follows. Section 2 describes our assumptions about the replicas and the network connecting them. Section 3 describes HQ, while Section 4 describes a number of optimizations and our new implementation of BFT. Section 5 presents analytic results for HQ, BFT, and Q/U performance in the absence of contention, and Section 6 provides performance results for HQ and BFT under various workloads. Section 7 discusses related work, and we conclude in Section 8. Appendix A presents a discussion of the liveness of our system and Appendix B presents the full protocol for contention resolution when we avoid the expense of digital signatures in the base protocol.

2 Model

The 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 server processes (or replicas). Client and server processes are classified as either correct or faulty. Correct processes are constrained to obey their specification, i.e., they follow the prescribed algorithms. Faulty processes may deviate arbitrarily from their specification: we assume a Byzantine failure model [8]. Note that faulty processes include those that fail benignly as well as those suffering from Byzantine failures.

We assume an asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, corrupt them, or deliver them out of order, and there are no known bounds on message delays or on the time to execute operations. We assume the network is fully connected, i.e., given a node identifier, any other node can (attempt to) contact the former directly by sending it a message.

For liveness, we require only that if a client keeps retransmitting a request to a correct server, the reply to that request will eventually be received, plus the conditions required for liveness of the BFT algorithm [2] that we use as a separate module.

We assume nodes can use unforgeable digital signatures to authenticate communication. We denote message m signed by node n as $\langle m \rangle_{\sigma_n}$. No node can send $\langle m \rangle_{\sigma_n}$ (either directly or as part of another message) on the network for any value of m, unless it is repeating a message that has been sent before or it knows $n's$ private key. We discuss how to avoid the use of computationally expensive digital signatures in Section 4. Message Authentication Codes (MACs) are used to establish secure communication between pairs of nodes, with the notation $\langle m \rangle_{\mu_{xy}}$ indicating a message authenticated using the symmetric key shared by x and y. We assume a trusted key distribution mechanism that provides each node with the public key of any other node in the system, thus allowing establishment of symmetric session keys for use in MACs.

We assume the existence of a collision-resistant hash function, h, such that 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 $h(m) = h(m')$.

To avoid replay attacks we tag certain messages with nonces that are signed in replies. We assume that when clients pick nonces they will not choose a repeated value.

3 HQ Replication

HQ is a state machine replication protocol that can handle arbitrary (deterministic) operations. We classify operations as *reads* and *writes*. (Note that the operations are not restricted to simple reads or writes of portions of the service state; the distinction made here is that read operations do not modify the service state whereas write operations do.) In the normal case of no failures and no contention, write operations require two phases to complete (we call the phases write-1 and write-2) while reads require just one phase. Each phase consists of the client issuing an RPC call to all replicas and collecting a quorum of replies.

The HQ protocol requires $3f + 1$ replicas to survive f failures and uses quorums of size $2f + 1$. It makes use of *certificates* to ensure that write operations are properly ordered. A certificate is a quorum of authenticated messages from different replicas all vouching for some fact. The purpose of the write-1 phase is to obtain a *timestamp* that determines the ordering of this write relative to others. Successful completion of this phase provides

the client with a certificate proving that it can execute its operation at timestamp t. The client then uses this certificate to convince replicas to execute its operation at this timestamp in the write-2 phase. A write concludes when $2f+1$ replicas have processed the write-2 phase request, and the client has collected the respective replies.

In the absence of contention, a client will obtain a usable certificate at the end of the write-1 phase and succeed in executing the write-2 phase. Progress is ensured in the presence of slow or failed clients by the writeBackWrite and writeBackRead operations, allowing other clients to complete phase 2 on their behalf. When there contention exists, however, a client may not obtain a usable write certificate, and in this case it asks the system to *resolve* the contention for the timestamp in question. Our contention resolution process uses BFT to order the contending operations. It guarantees

1. if the write-2 phase completed for an operation o at timestamp t, o will continue to be assigned to t.
2. if some client has obtained a certificate to run o at t, but o has not yet completed, o will run at some timestamp $\geq t$.

In the second case it is possible that some replicas have already acted on the write-2 request to run o at t and as a result of contention resolution, they may need to undo that activity (since o has been assigned a different timestamp). Therefore all replicas maintain a single backup state so that they can undo the last write they executed. However, this undo is not visible to end users, since they receive results only when the write-2 phase has completed, and in this case the operation retains its timestamp.

3.1 System Architecture

The system architecture is illustrated in Figure 1. Our code runs as proxies on the client and server machines: the application code at the client calls an operation on the client proxy, while the server code is invoked by the server proxy in response to client requests. The server code maintains replica state; it performs application operations and must also be able to undo the most recently received (but not yet completed) operation (to handle reordering in the presence of contention).

The replicas also run the BFT state machine replication protocol [2], which they use to resolve contention; note that BFT is not involved in the absence of contention.

3.2 Normal Case

We now present the details of our protocol for the case where there is no need to resolve contention; Section 3.3

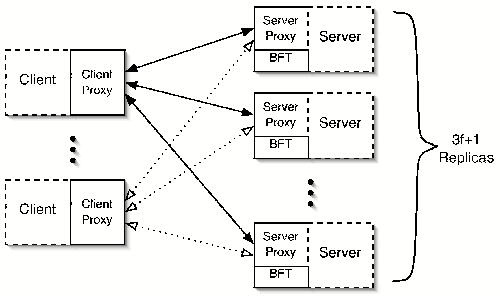


Figure 1: System Architecture

describes contention resolution. We present an unoptimized version of the protocol; optimizations are discusses in Section 4.

The system supports multiple objects. For now we assume that each operation concerns a single object; we discuss how to extend this to multi-object transactions in Section 3.6. Writers are allowed to modify different objects in parallel but are restricted to having only one operation outstanding on a particular object at a time. Writers number their requests on individual objects sequentially, which allows us to avoid executing modification operations more than once. A writer can query the system for information about its most recent write at any time.

A write certificate contains a quorum of *grants*, each of form $\langle cid,\ oid,\ op\#,\ h_{op},\ vs,\ ts,\ rid\rangle_{\sigma_r}$, where each grant is signed by its replica r with id rid. A grant states that the replica has granted the client with id cid the right to run the operation numbered $op\#$ whose hash is h_{op} on object oid at timestamp ts. A write certificate is valid if all the grants are from different replicas, are correctly authenticated, and are otherwise identical. We use the notation $c.cid$, $c.ts$, etc., to denote the corresponding components of write certificate c. The vs component is a *viewstamp*; it tracks the running of BFT to perform contention resolution. A certificate $C1$ is *later than* certificate $C2$ if $C1$'s viewstamp or timestamp is larger than that of $C2$. (A viewstamp is a pair consisting of the current BFT view number, plus the number assigned by BFT to the operation it executed most recently, with the obvious ordering.)

3.2.1 Processing at the Client

Write Protocol. As mentioned, writing is performed in two phases. In the first phase the writer obtains a certificate that allows it to execute the operation at a particular timestamp in the second phase.

Write-1 phase. The client sends a $\langle$WRITE-1, *cid, oid, op#, op*$\rangle_{\sigma_c}$ request to the replicas. The following replies

are possible:

- ⟨WRITE-1-OK, *grantTS, currentC*⟩, if the replica granted the next timestamp to this client.
- ⟨WRITE-1-REFUSED, *grantTS, cid, oid, op#, currentC*⟩$_{\mu_{cr}}$, if the replica granted the timestamp to some other client; the reply contains the grant to that client, plus the information about this client's request (to prevent replays).
- ⟨WRITE-2-ANS, *result, currentC, rid*⟩$_{\mu_{cr}}$, if the client's write has already been executed (this can happen if this client is slow and some other client performs the write for it – see step 2 below).

In a WRITE-1-OK or WRITE-1-REFUSED reply, *currentC* is the certificate for the latest write done at that replica.

The client discards invalid replies; it processes valid replies as follows:

1. If it receives a quorum of OKs for the same viewstamp and timestamp, it forms a write certificate from the grants in the replies and moves to the write-2 phase.
2. If it receives a quorum of refusals with the same viewstamp, timestamp, and hash, some other client has received a quorum of grants and should be executing a write phase 2. To facilitate progress in the presence of slow or failed clients, the client forms a certificate from these grants and performs the write followed by a repeat of its own WRITE-1 request: it sends a ⟨WRITEBACKWRITE, *writeC, w1*⟩ to the replicas, where $w1$ is a copy of its WRITE-1 request; replicas reply with their responses to the WRITE-1.
3. If it receives grants with different viewstamps or timestamps, it also performs a WRITEBACKWRITE, this time using the latest write certificate it received. This case can happen when some replica has not heard of an earlier write. Writebacks are sent only to the slow replicas.
4. If it receives a WRITE-2-ANS, it uses the certificate in the WRITE-2-ANS to move to phase 2. This case can happen if some other client performed its write (did step 2 above).
5. If it receives a quorum of responses containing grants with the same viewstamp and timestamp but otherwise different, it sends a RESOLVE request to the replicas; the handling of this request is discussed in Section 3.3. This situation indicates the possibility of write contention, The replies to the RESOLVE request are identical to replies to a WRITE-1 request, so the responses are handled as described in this section.

Write-2 phase. The client sends a ⟨WRITE-2, *writeC*⟩ request, where $writeC$ is the write certificate it obtained in phase 1. Then it waits for a quorum of valid matching responses of the form ⟨WRITE-2-ANS, *result, currentC, rid*⟩$_{\mu_{cr}}$; once it has these responses it returns to the calling application. It will receive this many matching responses unless there is contention; we discuss this case in Section 3.3.

Read Protocol. The client sends ⟨READ, *cid, oid, op, nonce*⟩$_{\mu_{cr}}$ requests to the replicas. The *nonce* is used to uniquely identify the request, allowing a reader to distinguish the respective reply from a replay. The response to this request has the form ⟨READ-ANS, *result, nonce, currentC, rid*⟩$_{\mu_{cr}}$.

The client waits for a quorum of valid matching replies and then returns the result to the calling application. If it receives replies with different viewstamps or timestamps, it sends a ⟨WRITEBACKREAD, *writeC, cid, oid, op, nonce*⟩$_{\mu_{cr}}$ to the (slow) replicas, requesting that they perform the write followed by the read. Here $writeC$ is the latest write certificate received in the replies. This case can occur when a write is running concurrently with the read.

3.2.2 Processing at the Replicas

Now we describe processing at the replicas in the absence of contention resolution. Each replica keeps the following information for each object:

- *currentC*, the certificate for the current state of the object.
- *grantTS*, a grant for the next timestamp, if one exists
- *ops*, a list of WRITE-1 requests that are currently under consideration (the request that was granted the next timestamp, and also requests that have been refused), plus the request executed most recently.
- *oldOps*, a table containing, for each client authorized to write, the *op#* of the most recently completed write request for that client together with the result and certificate sent in the WRITE-2-ANS reply to that request.
- *vs*, the current viewstamp (which advances each time the system does contention resolution).

A replica discards invalid requests (bad format, improperly signed, or invalid certificate). It processes valid requests as follows:

Read request ⟨READ, *cid, oid, op, nonce*⟩$_{\mu_{cr}}$. The replica does an upcall to the server code, passing it the *op*. When this call returns it sends the result to the client in a message ⟨READ-ANS, *result, nonce, currentC,*

rid$\rangle_{\mu_{cr}}$. The nonce is used to ensure that the answer is not a replay.

Write 1 request ⟨WRITE-1, *cid, oid, op#, op*⟩$_{\sigma_c}$. If $op\# < oldOps[cid].op\#$, the request is old and is discarded. If $op\# = oldOps[cid].op\#$, the replica returns a WRITE-2-ANS response containing the result and certificate stored in $oldOps[cid]$. If the request is stored in ops, the replica responds with its previous WRITE-1-OK or WRITE-1-REFUSED response. Otherwise the replica appends the request to ops. Then if $grantTS = null$, it sets $grantTS = \langle$ *c, oid, op#, h, vs, currentC.ts+1, rid*$\rangle_{\sigma_r}$, where h is the hash of ⟨ *cid, oid, op#, op*⟩ and replies ⟨WRITE-1-OK, *grantTS, currentC*⟩; otherwise it replies ⟨WRITE-1-REFUSED, *grantTS, cid, oid, op#, currentC*⟩$_{\mu_{cr}}$ (since some other client has been granted the timestamp).

Write 2 request ⟨WRITE-2, *writeC*⟩. Any node is permitted to run a WRITE-2 request; the meaning of the request depends only on the contained certificate rather than on the identity of the sender. The certificate identifies the client c that ran the write-1 phase and the oid and $op\#$ it requested. The replica uses the oldOps entry for c to identify old and duplicate write-2 requests; it discards old requests and returns the write-2 response stored in $oldOps[c]$ for duplicates.

If the request is new, the replica makes an upcall to the server code to execute the operation corresponding to the request. A replica can do the upcall to execute a WRITE-2 request only if it knows the operation corresponding to the request and it is up to date; in particular its $vs = writeC.vs$ and $currentC.ts = writeC.ts - 1$. If this condition isn't satisfied, it obtains the missing information from other replicas as discussed in Sections 3.3 and 3.4, and makes upcalls to perform earlier operations before executing the current operation.

When it receives the result of the upcall, the replica updates the information in the *oldOps* for c, sets *grantTS* to $null$, sets ops to contain just the request being executed, and sets *currentC* = *writeC*. Then it replies ⟨WRITE-2-ANS, *result, currentC, rid*⟩$_{\mu_{cr}}$.

WriteBackWrite and WriteBackRead. The replica performs the WRITE-2 request, but doesn't send the reply. Then it processes the READ or WRITE-1 and sends that response to the client.

3.3 Contention Resolution

Contention occurs when several clients are competing to write at the same timestamp. Clients notice contention when processing responses to a WRITE-1 request, specifically case (5) of this processing, where the client has received conflicting grants for the same viewstamp and timestamp. Conflicting grants normally arise because of contention but can also occur because a faulty client has sent different requests to different replicas.

In either case a client requests contention resolution by sending a ⟨RESOLVE, *conflictC, w1*⟩ request to the replicas, where $conflictC$ is a *conflict certificate* formed from the grants in the replies and $w1$ is the WRITE-1 request it sent that led to the conflict being discovered. The processing of this request orders one or more of the contending requests and performs those requests in that order; normally all contending requests will be completed.

To resolve contention we make use of the BFT state machine protocol [2], which is also running at the replicas. One of these replicas is acting as the BFT primary, and the server proxy code tracks this information, just like a client of BFT. However in our system, we use BFT only to reach agreement on a deterministic ordering of the conflicting updates.

3.3.1 Processing at Replicas

To handle contention, a replica has additional state:

- *conflictC*, either *null* or the conflict certificate that started the conflict resolution.
- *backupC*, containing the previous write certificate (for the write before the one that led to the $currentC$ certificate).
- *prev*, containing the previous information stored in *oldOps* for the client whose request was executed most recently.

A replica that is processing a resolve request has a non-null value in $conflictC$. Such a replica is *frozen*: it does not respond to client write and resolve requests, but instead delays this processing until conflict resolution is complete.

When a non-frozen replica receives a valid ⟨RESOLVE, *clientConflictC, w1*⟩ request, it proceeds as follows:

- If $currentC$ is later than $clientConflictC$, or if the viewstamps and timestamps match but the request has already been executed according to the replica's $oldOps$, the conflict has already been resolved (by contention resolution in the case where the viewstamp in the message is less than vs). The request is handled as a WRITE-1 request.
- Otherwise the replica stores $clientConflictC$ in $conflictC$ and adds $w1$ to ops if it is not already there. Then it sends a ⟨START, *conflictC, ops, currentC, grantTS*⟩$_{\sigma_r}$ message to the server proxy code running at the current primary of the BFT protocol.

When the server proxy code running at the primary receives a quorum of valid START messages (including

one from itself) it creates a BFT operation to resolve the conflict. The argument to this operation is the quorum of these START messages; call this $startQ$. Then it causes BFT to operate by passing the operation request to the BFT code running at its node. In other words, the server proxy becomes a client of BFT, invoking an operation on the BFT service implemented by the same replica set that implements the HQ service.

BFT runs in the normal way: the primary orders this operation relative to earlier requests to resolve contention and starts the BFT protocol to obtain agreement on this request and ordering. At the end of agreement each replica makes an upcall to the server proxy code, passing it $startQ$, along with the current viewstamp (which has advanced because of the running of BFT).

In response to the upcall, the server proxy code produces the new system state; now we describe this processing. In this discussion we will use the notation *startQ.currentC*, *startQ.ops*, etc., to denote the list of corresponding components of the START messages in $startQ$.

Producing the new state occurs as follows:

1. If $startQ$ doesn't contain a quorum of correctly signed START messages, the replica immediately returns from the upcall, without doing any processing. This can happen only if the primary is faulty. The replica makes a call to BFT requesting it to do a view change; when this call returns, it sends its START message to the new primary.
2. The replica determines whether $startQ.grantTS$ forms a certificate (i.e., it consists of a quorum of valid matching grants). It chooses the grant certificate if one exists, else the latest valid certificate in $startQ.currentC$; call this certificate C.
3. Next the replica determines whether it needs to undo the most recent operation that it performed prior to conflict resolution; this can happen if some client started phase 2 of a contending write and the replica had executed its WRITE-2 request, yet none of the replicas that contributed to $startQ$ knew about the WRITE-2 and some don't even know of a grant for that request. The replica can recognize the situation because $currentC$ is later than C. To undo, it makes an upcall to the server code, requesting the undo. Then it uses $prev$ to revert the state in $oldOps$ for the client that requested the undone operation, and sets $currentC$ to $backupC$.
4. Next the replica brings itself up to date by executing the operation identified by C, if it hasn't already done so. This processing may include executing even earlier operations, which it can obtain from other replicas if necessary, as discussed in Section 3.4. It executes the operations by making upcalls to the server code and updates its $oldOps$ to reflect the outcome of these executions. Note that all honest replicas will have identical $oldOps$ after this step.
5. Next the replica builds an ordered list L of operations that need to be executed. L contains all valid non-duplicate (according to its $oldOps$) requests in $startQ.ops$, except that the replica retains at most one operation per client; if a (faulty) client submitted multiple operations (different hashes), it selects one of these in a deterministic way, e.g., smallest hash. The operations in L are ordered in some deterministic way, e.g., based on the cid ordering of the clients that requested them.
6. The operations in L will be executed in the selected order, but first the replica needs to obtain certificates to support each execution. It updates vs to hold the viewstamp given as an argument of the upcall and sends a grant for each operation at its selected timestamp to all other replicas.
7. The replica waits to receive $2f + 1$ valid matching grants for each operation and uses them to form certificates. Then it executes the operations in L in the selected order by making upcalls, and updating ops, $oldOps$, $grantTS$ and $currentC$ (as in Write-2 processing) as these executions occur.
8. Finally the replica clears $conflictC$ and replies to the RESOLVE request that caused it to freeze (if there is one); this processing is like that of a WRITE-1 request (although most likely a WRITE2-ANS response will be sent).

Conflict resolution has no effect on the processing of WRITE-1 and READ request. However, to process requests that contain certificates (WRITE-2, RESOLVE, and also the write-back requests) the replica must be as up to date as the client with respect to contention resolution. The viewstamp conveys the needed information: if the viewstamp in the certificate in the request is greater than vs, the replica calls down to the BFT code at its node, requesting to get up to date. This call returns the $startQ$'s and viewstamps for all the BFT operations the replica was missing. The replica processes all of this information as described above; then it processes the request as described previously.

A bad primary might not act on START messages, leaving the system in a state where it is unable to make progress. To prevent this, a replica will broadcast the START message to all replicas if it doesn't receive the upcall in some time period; this will cause BFT to do a view-change and switch to a new primary if the primary is the problem. The broadcast is also useful to handle bad clients that send the RESOLVE request to just a few replicas.

3.3.2 Processing at Clients

The only impact of conflict resolution on client processing is that a WRITE-2-ANS response might contain a different certificate than the one sent in the WRITE-2 request; this can happen if contention resolution ran concurrently with the write 2 phase. To handle this case the client selects the latest certificate and uses it to redo the write-2 phase.

3.4 State Transfer

State transfer is required to bring slow replicas up to date so that they may execute more recent writes. A replica detects that it has missed some updates when it receives a valid certificate to execute a write at timestamp t, but has an existing value of *currentC.ts* smaller than $t-1$.

A simple but inefficient protocol for state transfer is to request state from all replicas, for each missing update up to $t-1$, and wait for $f+1$ matching replies. To avoid transferring the same update from multiple replicas we take an optimistic approach, retrieving a single full copy of updated state, while confirming the hash of the updates from the remaining replicas.

A replica requests state transfer from $f+1$ replicas, supplying a timestamp interval for the required updates. One replica is designated to return the updates, while f others send a hash over this partial log. Responses are sought from other replicas if the hashes don't agree with the partial log, or after a timeout. Since the partial log is likely to be considerably larger than f hashes, the cost of state transfer is essentially constant with respect to f.

To avoid transferring large partial logs, we propose regular system checkpoints to establish complete state at all replicas [2]. These reduce subsequent writeback cost and allow logs prior to the checkpoint to be discarded. To further minimize the cost of state transfer, the log records may be compressed, exploiting overwrites and application-specific semantics [7]; alternatively, state may be transferred in the form of differences or Merkle trees [15].

3.5 Correctness

This section presents a high-level correctness argument for the HQ protocol. We prove only the safety properties of the system, namely that we ensure that updates in the system are linearizable [6], in that the system behaves like a centralized implementation executing operations atomically one at a time. A discussion of liveness can be found in Appendix A.

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 ordered subsequently in contention resolution can have a quorum of $2f+1$ existing WRITE-2-ANS responses. This follows from the above.

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 in Appendix A) 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.

3.6 Transactions

This section describes how we extend our system to support transactions that affect multiple objects.

We extend the client WRITE-1 request so that now it can contain more than one *oid*; in addition it must provide an *op#* for each object. Thus we now have ⟨WRITE-1, *cid, oid1, op1#, ..., oidk, opk#, op*⟩$_{\sigma_c}$. We still restrict the client to one outstanding operation per object; this implies that if it performs a multi-object operation, it cannot perform operations on any of the objects used in that operation until it finishes. Note that *op* could be a sequence of operations, e.g., it consists of $op_1; ...; op_m$, as perceived by the server code at the application.

The requirement for correct execution of a transaction is that for each object it uses it must be executed at the same place in the order for that object at *all* replicas. Furthermore it must not be interleaved with other transactions. For example suppose one transaction consisted of $op1(o1); op2(o2)$ while a second consisted of $op3(o1); op4(o2)$; then the assignment of timestamps cannot be such that $o3$ happens after $o1$ while $o4$ happens before $o2$.

We achieve these conditions as follows.

- When a replica receives a valid new multi-object request, and it can grant this client the next timestamp for each object, it returns a *multi-grant* of the form ⟨cid, h, vs, olist⟩$_{\sigma_r}$, where $olist$ contains an entry ⟨oid, op#, ts⟩ for each object used by the multi-object request; otherwise it refuses, returning all outstanding grants for objects in the request. In either case it returns its most recent certificate for each requested object.

- A client can move to phase 2 if it receives a quorum of matching multi-grants. Otherwise it either does a WRITEBACKWRITE or requests contention resolution. The certificate in the WRITE-2 request contains a quorum of multi-grants; it is valid only if the multi-grants are identical.

- A replica processes a valid WRITE-2 request by making a single upcall to the application. Of course it does this only after getting up to date for each object used by the request. This allows the application to run the transaction atomically, at the right place in the order.

- To carry out a resolve request, a replica freezes for *all* objects in the request and performs conflict resolution for them simultaneously: Its START message contains information for each object identified in the RESOLVE request.

- When processing $startQ$ during contention resolution, a replica retains a most one valid request per client per object. It orders these requests in some deterministic way and sends grants to the other replicas; these will be multi-grants if some of these request are multi-object operations, and the timestamp for object o will be $o.currentC.ts + |L_o|$, where L_o is L restricted to requests that concern object o. It performs the operations in the selected order as soon as it obtains the $newC$ certificate.

4 Optimizations

There are a number of ways to improve the protocol just described. For example, the WRITE-2-ANS can contain the client op# instead of a certificate; the certificate is needed only if it differs from what the client sent in the request. In addition we don't send certificates in responses to WRITE-1 and READ requests, since these are used only to do writebacks, which aren't needed in the absence of contention and failures; instead, clients need to fetch the certificate from the replicas returning the largest timestamp before doing the writeback. Another useful optimization is to avoid sending multiple copies of results in responses to READ and WRITE-2 requests; instead, one replica sends the answer, while the others send only hashes, and the client accepts the answer if it matches the hashes. Yet another improvement is to provide grants optimistically in responses to WRITE-1 requests: if the replica is processing a valid WRITE-2 it can grant the next timestamp to the WRITE-1 even before this processing completes. (However, it *cannot* reply to the WRITE-2 until it has performed the operation.)

Below we describe two additional optimizations: early grants and avoiding signing. In addition we discuss preferred quorums, and our changes to BFT.

4.1 Early Grants

The conflict resolution strategy discussed in Section 3.3 requires an extra round of communication at the end of running BFT in order for replicas to obtain grants and build certificates.

We avoid this communication by producing the grants while running BFT. The BFT code at a replica executes an operation by making an upcall to the code running above it (the HQ server code in our system) once it has received a quorum of valid COMMIT messages. We modify these messages so that they now contain grants. This is done by modifying the BFT code so that prior to sending *commit* messages it does a MAKEGRANT upcall to

the server proxy code, passing it $startQ$ and the viewstamp that corresponds to the operation being executed. The server code determines the grants it would have sent in processing $startQ$ and returns them in its response; the BFT code then piggybacks the grants on the COMMIT message it sends to the other replicas.

When the BFT code has the quorum of valid COMMIT messages, it passes the grants it received in these messages to the server proxy code along with $startQ$ and the viewstamp. If none of the replicas that sent COMMIT messages is faulty, the grants will be exactly what is needed to make certificates. If some grants are bad, the replica carries out the post phase as described in Section 3.3.

The grants produced while running BFT could be collected by a malicious intruder or a bad replica. Furthermore, the BFT operation might not complete; this can happen if the BFT replicas carry out a view change, and fewer than $f + 1$ honest replicas had sent out their COMMIT messages prior to the view change. However, the malicious intruder can't make a certificate from grants collected during the running of a single aborted BFT operation, since there can be at most $2f$ of them, and it is unable to make a certificate from grants produced during the execution of different BFT operations because these grants contain different viewstamps.

4.2 Avoiding Signing

In Section 3, we assumed that grants and WRITE-1 requests were signed. Here we examine what happens when we switch instead to MACs (for WRITE-1 requests) and *authenticators* (for grants). An authenticator [2] is a vector of MACs with an entry for each replica; replicas create authenticators by having a secret key for each other replica and using it to create the MAC for the vector entry that corresponds to that other replica.

Authenticators and MACs work fine if there is no contention and no failures. Otherwise problems arise due to an important difference between signatures and authenticators: A signature that any good client or replica accepts as valid will be accepted as valid by all good clients and replicas; authenticators don't have this property. For example, when processing $startQ$ replicas determined the most recent valid certificate; because we assumed signatures, we could be sure that all honest replicas would make the same determination. Without signatures this won't be true, and therefore we need to handle things differently.

The only place where authenticators cause problems during non-contention processing is in the responses to WRITE-2 and writeback requests. In the approach described in Section 3.2, replicas drop bad WRITE-2 and writeback requests. This was reasonable when using signatures, since clients can avoid sending bad certificates. But clients are unable to tell whether authenticators are valid; they must rely on replicas to tell them.

Therefore we provide an additional response to WRITE-2 and writeback requests: the replica can send a WRITE-2-REFUSED response, containing a copy of the certificate, and signed by it. When a client receives such a response it requests contention resolution.

The main issue in contention resolution is determining the latest valid certificate in $startQ$. It doesn't work to just select the certificate with the largest timestamp, since it might be forged. Furthermore there might be two or more certificates for the same highest timestamp but different requests; the replicas need to determine which one is valid.

We solve these problems by doing some extra processing before running BFT. Here is a sketch of how it works; a full discussion can be found in Appendix B.

To solve the problem of conflicting certificates that propose the same timestamp but for different requests, the primary builds $startQ$ from START messages as before except that $startQ$ may contain more than a quorum of messages. The primary collects START messages until there is a subset $startQ_{sub}$ that contains no conflicting certificates. If two START messages propose conflicting certificates, neither is placed in $startQ_{sub}$; instead the primary adds another message to $startQ$ and repeats the analysis. It is safe for the primary to wait for an additional message because at least one of the conflicting messages came from a dishonest replica.

This step ensures that $startQ_{sub}$ contains at most one certificate per timestamp. It also guarantees that at least one certificate in $startQ_{sub}$ contains a timestamp greater than or equal to that of the most recently committed write operation because $startQ_{sub}$ contains at least $f + 1$ entries from non-faulty replicas, and therefore at least one of them supplies a late enough certificate.

The next step determines the latest valid certificate. This is accomplished by a voting phase in which replicas collect signed votes for certificates that are valid for them and send this information to the primary in signed ACCEPT messages; the details can be found in Appendix B. The primary collects a quorum of ACCEPT messages and includes these messages as an extra argument in the call to BFT to execute the operation. Voting can be avoided if the latest certificate was formed from $startQ.grantTS$ or proposed by at least $f + 1$ replicas.

This step retains valid certificates but discards forged certificates. Intuitively it works because replicas can only get votes for valid certificates.

When replicas process the upcall from BFT, they use the extra information to identify the latest certificate. An additional point is that when replicas create the set L of additional operations to be executed, they add an

operation to L only if it appears at least $f + 1$ times in $startQ.ops$. This test ensures that the operation is vouched for by at least one non-faulty replica, and thus avoids executing forged operations.

This scheme executes fewer requests than the approach discussed in Section 3.3. In particular, a write request that has already reached phase 2 will be executed in the scheme discussed in Section 3.3, but now it might not be (because it doesn't appear at least $f + 1$ times in $startQ.ops$). In this case when the WRITE-2 request is processed by a replica after contention resolution completes, the replica cannot honor the request. Instead it sends a WRITE-2-RETRY response containing a grant for the next timestamp, either for this client or some other client. When a client gets this response, it re-runs phase 1 to obtain a new certificate before retrying phase 2.

4.3 Preferred Quorums

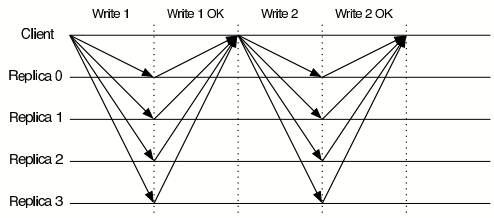
With preferred quorums, only a predetermined quorum of replicas carries out the protocol during fault-free periods. This technique is used in Q/U and is similar to the use of witnesses in Harp [10]. In addition to reducing cost, preferred quorums ensure that all client operations intersect at the same replicas, reducing the frequency of writebacks.

Since ultimately every replica must perform each operation, we have clients send the WRITE-1 request to all replicas. However, only replicas in the preferred quorum respond, the authenticators in these responses contain entries only for replicas in the preferred quorum, and only replicas in the preferred quorum participate in phase 2. If clients are unable to collect a quorum of responses, they switch to an unoptimized protocol using a larger group.

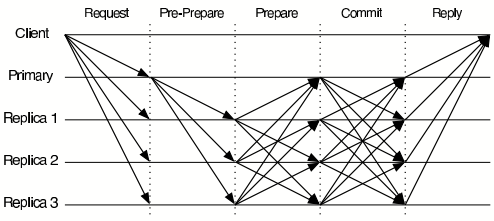
Replicas not in the preferred quorum need to periodically learn the current system state, in particular the timestamp of the most recently committed operation. This communication can be very lightweight, since only metadata and not client operations need be fetched.

4.4 BFT Improvements

The original implementation of BFT was optimized to perform well at small f, e.g., at $f = 2$. Our implementation is intended to scale as f increases. One main difference is that we use TCP instead of UDP, to avoid costly message loss in case of congestion at high f. The other is the use of MACs instead of authenticators in protocol messages. The original BFT used authenticators to allow the same message to be broadcast to all other replicas with a single operating system call, utilizing IP multicast if available. However, authenticators add linearly-scaling overhead to each message, with this extra cost becoming significant at high f in a non-broadcast medium.



(a) Quorum-based: HQ



(b) Agreement-based: BFT

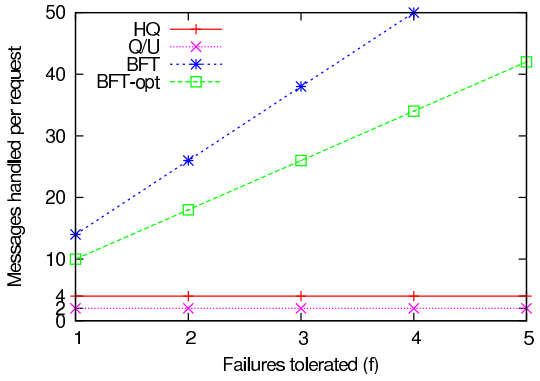
Figure 2: Protocol communication patterns.

Additionally, our implementation of BFT allows the use of preferred quorums.

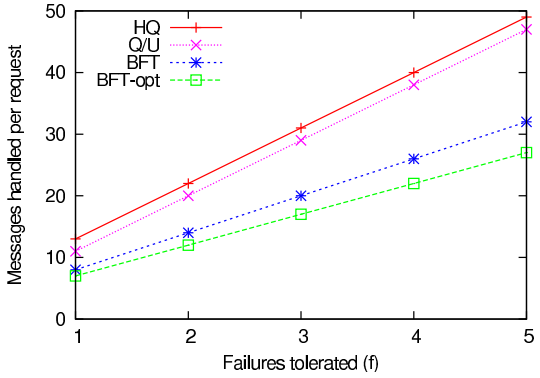
5 Analysis

Here we examine the performance characteristics of HQ, BFT, and Q/U analytically; experimental results can be found in Section 6. We focus on the cost of write operations since all three protocols offer one-phase read operations, and we expect similar performance in this case. We also focus on performance in the normal case of no failures and no contention. For both HQ and Q/U we use assume preferred quorums and MACs/authenticators. We show results for the original BFT algorithm (using authenticators and without preferred quorums), BFT-MACs (using MACs but not preferred quorums), and BFT-opt (using both MACs and preferred quorums). We assume the protocols use point-to-point communication.

Figure 2 shows the communication patterns for BFT and HQ; the communication pattern for Q/U is similar to the first round of HQ, with a larger number of replicas. Assuming that latency is dominated by the number of message delays needed to process a request, we can see that the latency of HQ is lower than that of BFT and the latency for Q/U is half of that for HQ. One point to note is that BFT can be optimized so that replicas reply to the client following the *prepare* phase, eliminating *commit*-phase latency in the absence of failures; with this optimization BFT can achieve the same latency as HQ. However, to amortize its quadratic communication costs, BFT employs batching, committing a group of operations as a single unit. This can lead to additional latency over a

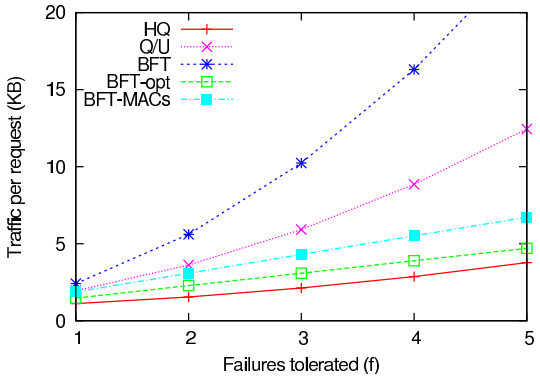


(a) Server

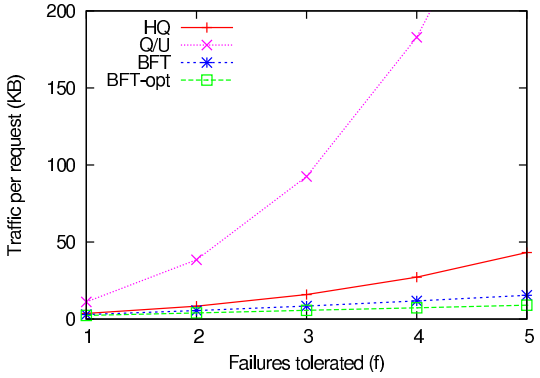


(b) Client

Figure 3: Total messages sent/received per write operation in BFT, Q/U, and HQ



(a) Server



(b) Client

Figure 4: Total traffic sent/received per write operation in BFT, Q/U, and HQ

quorum-based scheme.

Figure 3 shows the total number of messages required to carry out a write request in the three systems; the figure shows the load at both clients and servers. Consider first Figure 3a, which shows the load at servers. 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 runs through BFT, each replica must process $12f + 2$ messages. This is reduced to $8f + 2$ messages in BFT-opt by using preferred quorums.

Figure 3b shows the load at the client. Here we see that BFT-opt has the lowest cost, since a client just sends the request to the replicas and receives a quorum of responses. Q/U also requires one message exchange, but it has larger quorums (of size $4f+1$), for $9f+2$ messages. HQ has two message exchanges but uses quorums of size $2f + 1$; therefore the number of messages processed at the client, $9f + 4$, is similar in HQ to Q/U.

Figure 4 shows the total byte count of the messages processed to carry out a write request. This is computed using 20 byte SHA-1 digests [17] and HMAC authentication codes [16], 44 byte TCP/IP overhead, and a nominal request payload of 256 bytes. We analyze the fully optimized version of Q/U, using *compact timestamps* and *replica histories* pruned to the minimum number of two *candidates*. The results for BFT in Figure 4a show that our optimizations (MACs and preferred quorums) have a major impact on the byte count at replicas. The use of MACs 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.

Figure 4a also shows results for HQ and Q/U. In HQ the responses to the WRITE-1 request contains an authenticator and the WRITE-2 request contains a certificate, which grows quadratically with f. Q/U is similar: The response to a write returns what is effectively a grant (*replica history*), and these are combined to form a certificate (*object history set*), which is sent in the next write request. However, the grants in Q/U are considerably larger than those in HQ and also contain bigger

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 $f = 2x$.

Figure 4b shows the bytes required at the client. Here the load for BFT is low, since the client simply sends the request to all replicas and receives the response. The load for Q/U is the highest, owing to the quadratically growing certificates, larger grants and communication with approximately twice as many replicas.

6 Experimental Evaluation

This section provides performance results for HQ and BFT in the case of no failures. Following [1], we focus on performance of writes in a counter service supporting increment and fetch operations. The system supports multiple counter objects; each client request involves a single object and the client waits for a write to return before executing the subsequent request. In the non-contention experiments different clients use different objects; in the contention experiments a certain percentage of requests goes to a single shared object.

To allow meaningful comparisons of HQ and BFT, we produced new implementations of both, derived from a common C++ codebase. Communication is implemented 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 access required in executing each operation.

Our experiments ran on Emulab [20], 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 our experiments. Fedora Core 4 is installed on all machines, running Linux kernel 2.6.12.

Sixteen machines host a single replica each, providing support up to $f = 5$, with each of the remaining 50 machines hosting two clients. We vary the number of logical clients between 20 and 100 in each experiment, to obtain maximum possible throughput. We need a large number of clients to fully load the system because we limit clients 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 our plots. We report throughput; we observed batch size and protocol message count in our experiments and these results match closely to the analysis in Section 5.

We begin by evaluating performance when there is no contention: we examine maximum throughput in HQ and BFT, as well as their scalability as f grows. Throughput is CPU-bound in all experiments, hence this figure reflects message processing expense and cryptographic operations, along with kernel message handling overhead.

Figure 5 shows that the lower message count and fewer communication phases in HQ is reflected in higher throughput. 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.

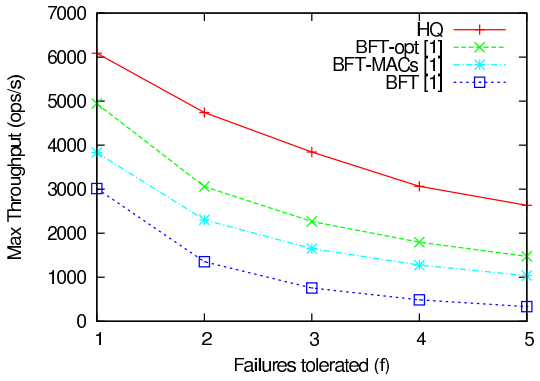


Figure 5: Maximum non-batched write throughput under varying f.

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

The results in Figure 5 don't tell the whole story. BFT can batch requests: the primary collects messages up to some bound, and then runs the protocol once per batch. Figure 6 shows that batching greatly improves BFT performance. The figure shows results for maximum batch sizes of 2, 5, and 10; in each case client requests may ac-

cumulate for up to 5ms at the primary, yielding observed batch sizes very close to the maximum.

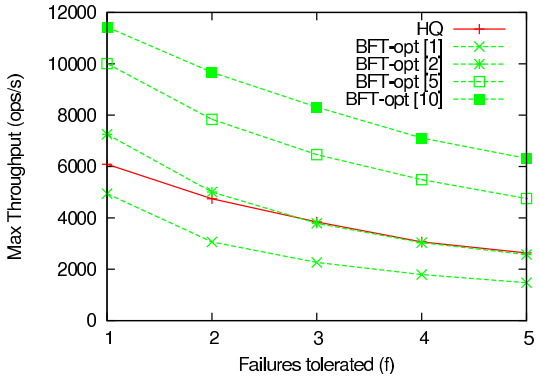


Figure 6: Effect of BFT batching on maximum write throughput.

Figure 7 shows the performance of HQ for $f = 1, ..., 5$ in the presence of write contention; in the figure *contention factor* is the fraction of writes executed on a single shared object. The figure shows that HQ performance degrades gracefully as contention increases. Performance reduction flattens significantly for high rates of write contention because multiple contending operations are ordered with a single round of BFT, achieving a degree of write batching. For example, at $f = 2$ this batching increases from an average of 3 operations ordered per round at contention factor 0.1 to 16 operations at contention factor 1.

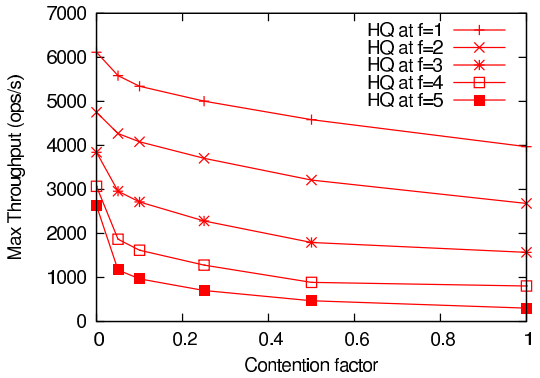


Figure 7: HQ throughput under increasing write contention.

7 Related Work

Byzantine quorum systems were introduced by Malkhi and Reiter [12]. The initial constructions were designed to handle only read and (blind) write operations. These were less powerful than state machine replication (implemented by our protocols) where the outcome of an operation can depend on the history of previously executed operations. Byzantine quorum protocols were later extended to support general object replication assuming benign clients (e.g. [13, 3]), and subsequently to support Byzantine clients but for larger non-blocking quorums [5]. Still more recent work showed how to handle Byzantine clients while using only $3f + 1$ replicas [11].

Recently, Abd-El-Malek et al. [1] demonstrated for the first time how to adapt a quorum protocol to implement state machine replication for multi-operation transactions with Byzantine clients. This is achieved with a combination of optimistic versioning and by having client requests store a history of previous operations they depend on, allowing the detection of conflicts in the ordering of operations (due to concurrency or slow replicas) and the retention of the correct version.

Our proposal builds on this work but reduces the number of replicas from $5f + 1$ to $3f + 1$. Our protocol does not require as many replicas owing to our mechanism for detecting and recovering from conflicting orderings of concurrent operations at different replicas. The Q/U protocols use a one-phase algorithm for writes; Abd-El-Malek et al show in their paper that their one-phase write protocol cannot run with fewer than $5f+1$ replicas (with quorums of size $4f + 1$). We use a two-phase write protocol, allowing us to require only $3f + 1$ replicas. A further difference of our work from Q/U is our use of BFT to order contending writes; this hybrid approach resolves contention much more efficiently than the approach used in Q/U, which resorts to an exponential backoff of concurrent writers that may lead to a substantial performance degradation.

In work done concurrently with that on Q/U, Martin and Alvisi [14] discuss the tradeoff 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 reaching agreement in two communication steps and they present a replica-based algorithm that shows this lower bound is tight.

Earlier proposals for Byzantine fault tolerant state-machine replication (e.g., Rampart [18] and BFT [2]) relied on inter-replica communication, instead of client-controlled, quorum-based protocols, to serialize requests. These protocols employ $3f+1$ replicas, and have quadratic communication costs in the normal case, since each operation involves a series of rounds where each replica sends a message to all remaining replicas, stating their agreement on an ordering that was proposed by a primary replica. An important optimization decouples the agreement from request execution [21] reducing the

number of the more expensive storage replicas to $2f + 1$ but still retaining the quadratic communication costs.

8 Conclusions

This paper presents HQ, a new protocol for Byzantine-fault-tolerant state-machine replication. HQ is a quorum based protocol that is able to run arbitrary operations. It reduces the required number of replicas from the $5f + 1$ needed in earlier work (Q/U) to the minimum of $3f + 1$ by using a two-phase instead of a one-phase write protocol.

Additionally we present a new way of handling contention in quorum-based protocols: we use BFT. Thus we propose a hybrid approach in which operations normally run optimistically, but a pessimistic approach is used when there is contention. The hybrid approach can be used broadly; for example it could be used in Q/U to handle contention, where BFT would only need to run at a predetermined subset of $3f + 1$ replicas.

We also presented a new implementation of BFT that was developed to scale with f.

Based on our analytic and performance results, we believe the following points are valid:

- In the region we studied (up to $f = 5$), if contention is low and low latency is the main issue, then if it is acceptable to use $5f + 1$ replicas, Q/U is the best choice else HQ is best since it outperforms BFT with a batch size of 1.

- Otherwise, BFT is the best choice in this region: it can handle high contention workloads, and it can beat the throughput of both HQ and Q/U through its use of batching.

- Outside of this region, we expect HQ will scale best. Our results show that as f grows, HQ's throughput decreases more slowly than Q/U's (because of the latter's larger messages and processing costs) and BFT's (where eventually batching cannot compensate for the quadratic number of messages).

9 Acknowledgments

We thank the anonymous reviewers and our shepherd Mema Roussopoulos for their valuable feedback and the developers of Q/U for their cooperation. We also thank the supporters of Emulab, which was absolutely crucial to our ability to run experiments.

This research was supported by NSF ITR grant CNS-0428107 and by T-Party, a joint program between MIT and Quanta Computer Inc., Taiwan.

References

[1] ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., REITER, M. K., AND WYLIE, J. J. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 59–74.

[2] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems 20*, 4 (Nov. 2002), 398–461.

[3] CHOCKLER, G., MALKHI, D., AND REITER, M. Backoff protocols for distributed mutual exclusion and ordering. In *Proc. of the IEEE International Conference on Distributed Computing Systems* (2001).

[4] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. 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, Nov. 2006).

[5] FRY, C., AND REITER, M. Nested objects in a byzantine Quorum-replicated System. In *Proc. of the IEEE Symposium on Reliable Distributed Systems* (2004).

[6] HERLIHY, M. P., AND WING, J. M. Axioms for Concurrent Objects. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages* (1987).

[7] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar Conference Center, Pacific Grove, CA., Oct. 1991), pp. 213–225.

[8] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

[9] LAMPORT, L. L. The implementation of reliable distributed multiprocess systems. *Computer Networks 2* (1978), 95–114.

[10] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles* (Pacific Grove, California, 1991), pp. 226–238.

[11] LISKOV, B., AND RODRIGUES, R. Byzantine clients rendered harmless. Tech. Rep. MIT-LCS-TR-994 and INESC-ID TR-10-2005, July 2005.

[12] MALKHI, D., AND REITER, M. Byzantine Quorum Systems. *Journal of Distributed Computing 11*, 4 (1998), 203–213.

[13] MALKHI, D., AND REITER, M. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering 12*, 2 (Apr. 2000), 187–202.

[14] MARTIN, J.-P., AND ALVISI, L. Fast byzantine consensus. In *International Conference on Dependable Systems and Networks* (2005), IEEE, pp. 402–411.

[15] MERKLE, R. C. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - Crypto'87*, C. Pomerance, Ed., no. 293 in Lecture Notes in Computer Science. Springer-Verlag, 1987, pp. 369–378.

[16] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Fips 198: The keyed-hash message authentication code (hmac), March 2002.

[17] NATIONAL INSTITUTE OF STANDARDS AND TECNOLOGY. Fips 180-2: Secure hash standard, August 2002.

[18] REITER, M. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)* (1995), 99–110.

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

[20] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[21] YIN, J., MARTIN, J., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Oct. 2003).

A Liveness

This section presents arguments that our approach is live. 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 [2].

The argument for liveness is as follows:

- When there is no contention, client requests execute in a number of phases that is bounded by a constant. 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.

- 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.

We note that while providing liveness, our system does not offer any guarantees on *fairness*. 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, as opposed to primary-driven agreement. In practice we ensure that competing clients get serviced whenever contention is detected and resolved using BFT.

B Full Contention Resolution Protocol

This section follows from Section 4.2 and describes our protocol for doing contention resolution when we use authenticators in our base protocol. Note that we continue to use signatures when performing contention resolution.

The protocol starts when one or more replicas receives a valid RESOLVE request. These replicas send $\langle$START, *conflictC, ops, currentC, grantTS,* $vs\rangle_{\sigma_r}$ messages to the primary (and to all replicas if they don't receive the corresponding upcall in some time period). Here vs is the current viewstamp at the replica (which is recorded as another component of 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. (When using signatures, at least one request will be executed at a timestamp containing the new viewstamp as part of running conflict resolution, but this may not happen when using authenticators.)

The primary collects these messages and checks them for validity. In particular it discards any START message containing a $currentC$ or $grantTS$ whose request is not present in ops.

Then the primary processes the messages and runs BFT as discussed below.

Step 1. The first problem we need to solve is that it's 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, and informs the replica that it needs to move to the new view.

Thus the only case of concern is the second one, which occurs when one of the replicas is lying. This was not a problem using when using signatures, since the signatures themselves were sufficient in indicating an invalid certificate. When using authenticators however, we have no guarantee that an authenticator appearing valid to a given replica will appear valid for others, and hence cannot rely on such checks.

Therefore we use an alternative technique. As in our base protocol, the primary collects all START messages into $startQ$, but it also collects non-conflicting messages into a set $startQ_{sub}$. 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, it doesn't add the new message and also removes the existing conflicting message from $startQ_{sub}$; 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.

This step terminates when $|startQ_{sub}| + k = 2f + 1$, where $k = |startQ - startQ_{sub}|/2$, the number of conflicting pairs. At this point $startQ_{sub}$ contains at least $f + 1$ entries from honest replicas and there are no conflicting proposals for current certificates among the entries in $startQ_{sub}$

Step 2. The primary now has a collection of at least $2f + 1$ START messages in $startQ$. If the latest certificate was formed from $startQ.grantTS$ or proposed by at least $f + 1$ replicas, the primary can immediately run BFT as discussed in Section 3.3.

Otherwise, it isn't clear which certificate is the latest, even considering just the certifications in $startQ_{sub}$, because it's possible that the certificate with the latest timestamp was proposed by a liar. To sort this out, the replicas carry out the following protocol.

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

2. Each replica r computes $startQ_{sub}$. Then it chooses all certificates C from $startQ_{sub}$ with timestamps in the range $[currentC.ts - 1, currentC.ts + 1]$, encompassing the cases where it is behind, 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 is valid for the replica or the replica has C's request in its $ops - curr$, where $curr$ is the request indicated in $currentC$ (recall that ops contains the new requests plus the one executed to reach $currentC$).

   Thus at this point the replica has between zero and three *candidate* certificates.

3. Each replica creates *votes* for all candidates. A vote is a pair $\langle ts, h\rangle$ where h is the hash of the request at timestamp $ts - 1$; note that this implies that each replica must retain an additional old certificate for the request at $currentC.ts - 2$ so that it can vote for a candidate at $currentC.ts - 1$. Then if it has any votes to send, it sends $\langle$OK, *V, $h_c\rangle_{\sigma_r}$* 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 $currentC.ts + 1$ and that certificate isn't valid for the replica, it is removed from the set of candidates.

5. If the replica has a nonempty set of candidates, it waits for $f + 1$ *matching* OK messages for the timestamp of the 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 it 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.

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. Then it calls BFT to run a CHECKEDRESOLVE operation, with $startQ$ and the collection of ACCEPT messages as arguments. Again the entries in $startQ$ are in the order it processed them.

Step 3. The replicas process the upcall from BFT as described in Section 3.3, except that in the case of a CHECKEDRESOLVE operation they use the ACCEPT messages to determine the latest certificate C: this is the certificate in $startQ_{sub}$ that contains the largest timestamp mentioned in one of the ACCEPT messages that contains a nonempty O.

If the replica is out of date, it must first perform state transfer to obtain the missing requests. If it is processing a RESOLVE upcall, it fetches state up to $C.ts - 1$ as discussed in Section 3.3. However, if it is processing a CHECKEDRESOLVE upcall, it uses the extra 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 $startQ.ops$.

B.1 Correctness

In this section we discuss the safety and liveness of the protocol given previously. As in Section 3.5 and Appendix A we assume that if replicas repeatedly send messages, they will eventually be delivered; we also assume there are at most f faulty replicas.

We begin by stating a few lemmas about Step 1.

- Lemma 1. $startQ_{sub}$ contains no conflicting proposals. This is obvious, by construction.
- Lemma 2. The set $startQ_{rejects} = startQ - startQ_{sub}$ contains at least $k = |startQ_{rejects}|/2$ messages from faulty replicas. This follows because 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 pair of conflicting messages is a liar.
- Lemma 3. The set $startQ_{rejects}$ contains at most k messages from honest replicas. This follows directly from Lemma 2.
- Lemma 4. When Step 1 terminates, $startQ_{sub}$ contains at least $f + 1$ messages from honest replicas. This can be shown as follows. When Step 1 terminates $|startQ_{sub}| = 2f + 1 - k = (f + 1) + (f - k)$. From Lemma 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.

B.1.1 Liveness

There are two places where we might have additional concerns about liveness (over those addressed in Appendix A) – the termination of Steps 1 and 2.

Step 1 terminates because whenever the primary waits for another START message we are certain there is at least one more non-faulty replica to send that message. The termination condition for this step is $|startQ_{sub}| + k = 2f + 1$. By Lemma 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 $2f+1$. Therefore it is 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 terminates because all honest nodes will send valid accept messages and therefore the primary will receive the $2f + 1$ valid ACCEPT messages needed for termination. If the honest replica has no candidate certificate where it is waiting for votes, it sends the ACCEPT message immediately. The 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. Therefore the replica that is waiting for votes will receive them and be able to send its ACCEPT message to the primary.

Note that it is important that replicas vote for several certificates; this handles the case where some of the honest replicas that processed the last committed request are ahead of the others because they are processing a write phase 2 for a later request. It is also important that replicas accept matching votes for a later certificate; this handles the case of an honest replica that is behind and is waiting for votes for a certificate with a timestamp earlier than that of the most recently committed request.

B.1.2 Safety

The correctness condition we must satisfy has two parts:

- *Commitment.* The latest certificate, C, selected by the protocol must be such that $C.ts \geq t$, 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.

Commitment. We satisfy the commitment condition because certificates from requests in $startQ_{sub}$ will include a late enough certificate, and furthermore, the selection process in Step 2 will select a late enough timestamp.

In Step 1 the concern is that as many as half of the messages in $startQ_{rejects}$ might be from honest replicas and as a result we might not have a message from an honest replica that knows about the most recently committed request in $startQ_{sub}$. However, by Lemma 4 we know that when Step 1 terminates, $startQ_{sub}$ contains at least $f + 1$ messages from honest replicas. Therefore it has a non-empty intersection with the set of at least $f + 1$ honest replicas that processed the write phase 2 message for the most recently committed request. Since that replica is honest, it will propose a certificate at least as late as the most recently committed request.

In Step 2, the honest replicas that know about the most recently committed request will succeed in collecting votes for it or for a higher request. Furthermore the primary will wait to receive an ACCEPT message from at least one of them, since there are $f + 1$ of them and the primary needs to receive $2f + 1$ ACCEPT messages. Therefore the latest certificate C will be sufficiently recent.

Validity. The request in the latest certificate C is certain to be valid because either C was 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 for it. In all these cases we can be sure that at least one honest replica vouches for C, which is sufficient to guarantee that the request in C is valid.

Furthermore all requests assigned timestamps greater than $C.ts$ are valid since they must appear at least $f + 1$ times in ops. So the only concern is for requests assigned timestamps less than $C.ts$. Furthermore there is no concern for requests with timestamps less than or equal to t, since these have already committed. So we are only concerned with requests assigned timestamps that are greater than t and less than $C.ts$.

In the case where C was selected without considering ACCEPT messages, there are no such requests: in this case either $C.ts = t$ or $C.ts = t + 1$.

However, if C is selected by considering ACCEPT messages, it is possible that $C.ts > t + 1$. In this case C is invalid and has been proposed by a faulty replica. However, that replica (or some other faulty replica) has succeeded in getting at least one non-faulty replica to vote for C. This implies two things. First, the request in C is valid, since a non-faulty replica will vote for it only in this case. Second, $C.ts \leq t + 2$ because the current certificate at a non-faulty replica cannot be more than one ahead of the most recently committed request and a non-faulty replica will not vote for a certificate that is more than one ahead of its $currentC$.

When $C.ts = t + 2$, the request selected to run at $t+1$ is the one whose hash is included in the votes for C. Since one of these votes comes from an honest replica, we can be sure that the request is valid.