*h*BFT: Speculative Byzantine Fault Tolerance With Minimum Cost

Sisi Duan, Sean Peisert, *Senior Member, IEEE,* and Karl N. Levitt

Abstract—We present *h*BFT, a hybrid, Byzantine fault-tolerant, replicated state machine protocol with optimal resilience. Under normal circumstances, *h*BFT uses speculation, i.e., replicas directly adopt the order from the primary and send replies to the clients. As in prior work such as Zyzzyva, when replicas are out of order, clients can detect the inconsistency and help replicas converge on the total ordering. However, we take a different approach than previous work that has four distinct benefits: it requires many fewer cryptographic operations, it moves critical jobs to the clients with no additional costs, faulty clients can be detected and identified, and performance in the presence of client participation will not degrade as long as the primary is correct. The correctness is guaranteed by a three-phase checkpoint subprotocol similar to PBFT, which is tailored to our needs. The protocol is triggered by the primary when a certain number of requests are executed or by clients when they detect an inconsistency.

Index Terms—distributed systems, client/server, fault tolerance, state machine replication

1 Introduction

As distributed systems develop and grow in size, Byzantine failures generated by malicious attacks, and software and hardware errors must be tolerated. Byzantine agreement protocols are attractive because they enhance reliability of replicated services in the presence of arbitrary failures. However, Byzantine protocols come at a cost of high overhead of messages and cryptographic operations. Therefore, protocols that can reduce overhead can be attractive building blocks.

A number of existing protocols also reduce overhead on Byzantine agreement by moving some critical jobs to clients [13, 17, 19, 21, 33, 34]. But these protocols come with trade-offs that we seek to avoid. Specifically, while they all provide better fault-free cases and reduce the message complexity, they sacrifice the performance of normal cases and may even decrease the performance of fault-free cases. For instance, the Zyzzyva [21] protocol is able to use roughly half of the amount of messages and cryptographic operations that PBFT [7] requires. However, Zyzzyva's performance can be even worse than PBFT if at least one backup fails. Additionally, these protocols simplify the design by involving clients in the agreement. However, they all require clients to be correct in order to achieve correctness.

Therefore, our motivation for developing a new protocol is to improve performance over PBFT without being encumbered by some of these trade-offs. Specifically, we have three key goals: first, we wish to be able to show how critical jobs can be moved to the clients without additional costs. Second, we wish to tolerate Byzantine faulty clients. Third, we define the notion of *normal case*, which means the primary is correct and the number of faulty backups does not exceed the threshold. We wish to provide better performance for both fault-free cases and normal cases.

This paper presents *h*BFT, a leader-based protocol that uses speculation to reduce the cost of Byzantine agreement, while also maintaining optimal resilience, utilizing $n \geq 3f+1$ replicas to tolerate f failures. *h*BFT satisfies all of our stated goals. To accomplish this, *h*BFT employs several techniques. It uses speculation: backups speculatively execute requests ordered by the primary as well as replies to the clients. As a result, correct replicas may be temporarily inconsistent. *h*BFT employs a three-phase PBFT-like checkpoint subprotocol for both garbage collection and contention resolution. The checkpoint subprotocol can be triggered by the replicas when they execute a certain number of operations, or by clients when they detect the divergence of replies. In this way replicas are able to detect any inconsistency through internal message exchanges. Even though the three-phase protocol is expensive, it is not triggered frequently. Eventually *h*BFT can ensure the total ordering of requests for all correct replicas with very low cost.

1.1 Motivation

Our goal for *h*BFT is to offer better performance by moving some critical jobs to the clients while minimizing side effects that can actually reduce performance in many cases in previous work [17, 21, 33, 34].

First, *h*BFT moves some critical jobs to the clients without additional cost. Moving critical jobs to the clients is effective in simplifying the design and reducing message complexity, partly because replicas do

S. Duan and K. N. Levitt are with the Department of Computer Science, University of California, Davis, CA. E-mail: {sduan,levitt}@cs.ucdavis.edu
S. Peisert is with the Department of Computer Science, University of California, Davis, CA and with Lawrence Berkeley National Laboratory, Berkeley, CA. E-mail: peisert@cs.ucdavis.edu

not need to run expensive protocols to establish the order for every request. Nevertheless, it does not necessarily make protocols more practical. Indeed, it may sacrifice performance in normal and even fault-free cases. For instance., the output commit in Zyzzyva slows both. *h*BFT achieves a simplified design and better performance for both fault-free and normal cases.

Second, *h*BFT can tolerate an unlimited number of faulty clients. Previous protocols all rely on the correctness of clients. However, Byzantine clients can dramatically decrease performance. For instance, in the protocols that switch between subprotocols [17, 33, 34] (called abstracts in [17]), a faulty client can stay silent when it detects the inconsistency. Even if the next client is correct and makes the protocol switch to another subprotocol, replicas are still inconsistent because of this "faulty request." Similarly, in Zyzzyva, faulty clients can stay silent when they are supposed to send a commit certificate to make all correct replicas converge. Faulty primaries in this case can not be detected, eventually leading to inconsistencies of replica states. Faulty clients can also intentionally send commit certificates to all replicas even if they receives $3f+1$ matching messages, which decreases the overall performance.

Third, *h*BFT has the same operations for both fault-free and normal cases. This shows that in leader-based protocols, when the primary is correct, all the requests are totally ordered by all correct replicas. Previous protocols all achieve impressive performance in fault-free cases while they employ different operations when failure occurs, resulting in lower performance. Although Zyzzyva5 [21] makes the faulty cases faster, it requires $5f+1$ replicas to tolerate f failures. In *h*BFT, we achieve better performance in both normal fault-free and normal cases using $3f+1$ replicas.

2 Related Work

Fig. 1 compares several features for *normal cases* between BFT protocols, a selection of which are plotted in Fig. 9. We provide the values for fault-free cases in the caption if they are different from normal cases. The table is constructed based on the models to tolerate f failures. As mentioned in previous work [21, 23, 27], Byzantine fault tolerant state machine replication protocols are known to have lower bounds. A protocol obtains optimal resilience if it uses $3f+1$ replicas to tolerate f failures. In addition, throughput and latency are measured through the number of cryptographic operations of the primary and one-way latencies. Although 2 is considered the lower bound of the cryptographic operations, it is not clear that this lower bound is achievable. However, with batching, the lower bound can be approached under high concurrency. On the other hand, 2 one-way latencies, achieved by Q/U [1] for instance, is considered the lower bound under low concurrency while 3 is the lower bound under high concurrency. Compared to other known, prior work, *h*BFT uses $3f+1$ replicas, the throughput approaches 2 under high concurrency, and achieves 3 one-way latencies. In summary, *h*BFT achieves optimal resilience and the lower bound for almost every feature under high concurrency.

Most current practical Byzantine fault tolerant protocols are developed based on PBFT [7], which is a three phase leader-based protocol. Subsequent work focus either on increasing the number of faults systems can tolerate or on improving performance. There are trade-offs between the two. For instance, Fab [27] is a two-phase protocol that achieves better performance by requiring at least $5f+1$ replicas in total to tolerate f failures. The checkpoint protocol of *h*BFT uses a tailored PBFT scheme, since it can guarantee correctness, but is too expensive to be used for fault-free and normal cases when the primary is correct.

Several protocols [17, 19, 21, 33, 34] move some critical jobs to the clients to improve performance. Zyzzyva and its variant [19] move output commit to the clients to reduce message complexity in fault-free cases. Other protocols [17, 33, 34] move the job of switching of subprotocols to the clients. When one subprotocol aborts, the protocol will switch to another. *h*BFT also switches between normal case operation and the checkpoint subprotocol. However, *h*BFT does not order any single request using the checkpoint subprotocol. Instead, it is used only for contention resolving and garbage collection. Clients can facilitate the progress in *h*BFT but clients do not need to provide any "proof" to replicas.

Byzantine quorum systems [1, 26] tolerate Byzantine faults under low concurrency. HQ [13] is a hybrid quorum and Byzantine agreement protocol that also uses a PBFT-like subprotocol to resolve contention. Compared to HQ, *h*BFT does not have an additional garbage collection scheme and it works well under high concurrency.

3 System Model

We consider a distributed system that tolerates a maximum of f faulty replicas and an unlimited number of faulty clients using $3f+1$ replicas. We consider the Byzantine fault tolerant replication problem, where faulty replicas and clients behave arbitrarily. In addition, we assume independent node failures, which can be obtained through techniques such as N-version programming [28].

Safety, which means requests are totally ordered by correct replicas, must hold in any asynchronous system using state machine replication, where messages can be delayed, dropped or delivered out of order. Liveness, which means correct clients eventually receive replies to their requests, is ensured assuming partial synchrony [15]: synchrony holds only

		PBFT [7]	Q/U [1]	HQ [13]	FaB [27]	Zyzzyva [21]	*h*BFT
Cost	Total replicas	$3f+1$	$5f+1$	$3f+1$	$5f+1$	$3f+1$	$3f+1$
Throughput (MAC ops/request)	Primary	$2+\frac{8f}{b}$	$2+8f^{\parallel}$	$4+4f^{\parallel}$	$1+\frac{5f}{b}$	$4+5f+\frac{3f}{b}^{\dagger}$	$2+\frac{3f}{b}$
	Backup	$2+\frac{8f+1}{b}$	$2+8f^{\parallel}$	$4+4f^{\parallel}$	$1+\frac{2f+2}{b}$	$4+5f+\frac{1}{b}^{*}$	$2+\frac{3f}{b}$
	Client	$2+4f$	$2+8f$	$4+4f$	$1+5f$	$4+10f^{\ddagger}$	$2+6f$
One-way Latencies	Critical path	4	2	4	3	$5^{\S}$	3
Works well on concurrency?		Yes	No	No	Yes	Yes	Yes
Handle faulty clients?		Yes	No	No	No	No	Yes

Fig. 1. Comparison of BFT protocols in normal cases tolerating f faults and using batch size b. †Fault-free cases: $2+\frac{3f}{b}$. *Fault-free cases: $2+\frac{1}{b}$. ‡Fault-free cases: $2+6f$. §Fault-free cases: 3. $^{\parallel}$Q/U and HQ are leader-free quorum systems that do not differentiate primary and backups.

after some unknown global stabilization time, but the bounds on communication and processing delays are themselves unknown.

Operations are executed in an atomic broadcast model, where correct replicas agree on the set of requests and the order of them. In the description that follows, when we refer to *fault-free cases*, we mean there are no replica failures, and when we refer to *normal cases*, we mean the primary is correct and the number of faulty backups is between 1 and f.

We use digital signatures, message authentication codes (MACs), and message digests to prevent spoofing and to detect corrupted messages. For a message m, $\langle m \rangle_i$ denotes the message with digital signature signed by replica p_i, $D(m)$ denotes the message digest, and $\langle m \rangle$ denotes the message with MAC $\mu_{i,j}(m)$. The MAC $\mu_{i,j}(m)$ is generated using secret key shared by replica p_i and p_j.

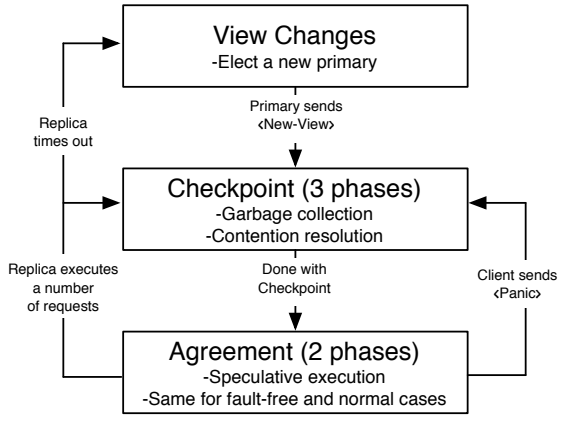


Fig. 2. Layered Structure of *h*BFT.

4 The *h*BFT Protocol

The *h*BFT protocol is a hybrid, replicated state machine protocol. It includes four major components: (1) agreement, (2) checkpoint, (3) view change, and (4) client suspicion. As illustrated in Fig. 2, we employ a simple agreement protocol for fault-free and normal cases, and use a three-phase checkpoint subprotocol for contention resolution and garbage collection. The checkpoint subprotocol can be triggered by replicas when they execute a certain number of requests or by clients if they detect divergence of replies. The view change subprotocol ensures liveness of the system and can coordinate the change of the primary. View changes can occur during normal operations or in the checkpoint subprotocol. In both cases, the new primary initializes a checkpoint subprotocol immediately and resumes the agreement protocol until a checkpoint becomes stable. The client suspicion subprotocol prevents faulty clients from attacking the system.

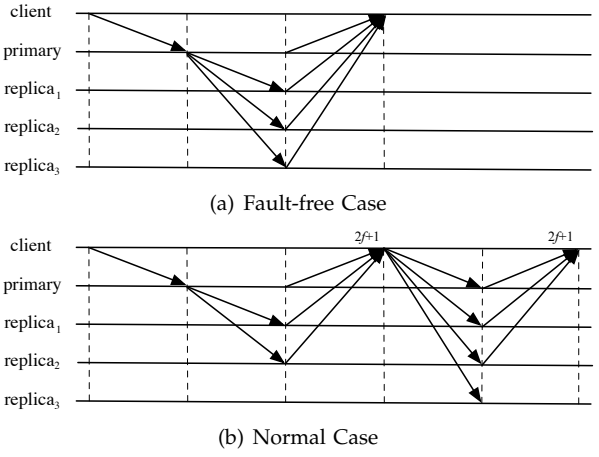


Fig. 3. Fault-free and normal cases of Zyzzyva.

Why another speculative BFT protocol?

*h*BFT uses speculation but overcomes some that problems Zyzzyva experiences. Zyzzyva [21] also uses speculation and moves output commit to the clients to enhance the performance. If we replace digital signatures with MACs and batch concurrent requests in Zyzzyva, the performance decreases in normal cases and even fault-free cases. Fig. 3 illustrates the behavior of Zyzzyva [21]. Replicas speculatively execute the requests and respond to the client. The client collects $3f+1$ matching responses to complete the request. If the client receives between $2f+1$ and $3f$ matching responses, it sends a commit certificate to all replicas,

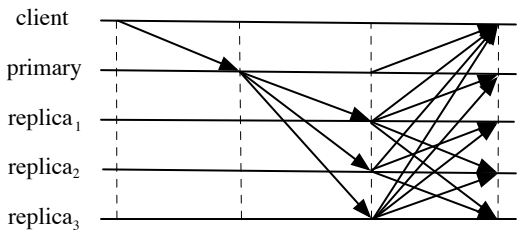


Fig. 4. The agreement protocol

which contains the response with $2f + 1$ signatures. This helps replicas converge on the total ordering. However, a commit certificate must be verified by every other replica, which causes computing overhead for both clients and replicas. The use of MACs instead of digital signatures makes Zyzzyva perform even worse than PBFT under certain configurations.[1] For a reply message r by replica p_i, $\langle r', \mu_{i,c}(r')\rangle$ must be sent to the client, where $r' = \langle r, \mu_{i,1}(r), \mu_{i,2}(r) \cdots \mu_{i,n}(r)\rangle$ and $\mu_{x,y}(r)$ denotes the MAC generated using the secret key shared by p_x and p_y. Therefore, every replica must include $3f + 1$ MACs for every reply message (compared with 1 if digital signatures are used) and performance is dramatically degraded. Assuming b is the batch size, the primary must perform $4 + 5f + \frac{3f}{b}$ MACs in normal cases, which is even worse than the $2 + \frac{8f}{b}$ MACs for PBFT for some b and f. Thus in hBFT, we seek to avoid this problem.

4.1 Agreement Protocol

The agreement protocol orders requests for execution by replicas. The algorithms of the agreement protocol for the primary, backups, and clients are defined in Algorithm 1 to Algorithm 3. As illustrated in Fig. 4, a client c invokes the operation by sending a $m = \langle \text{Request}, o, t, c\rangle_c$ to all replicas where o is the operation, t is the local timestamp. Upon receiving a request, as shown in Algorithm 1, the primary p_i assigns a sequence number seq and then sends out a $\langle \text{Prepare}, v, seq, D(m), m, c\rangle$ to all replicas, where v is the view number and $D(m)$ is the message digest.

A $\langle$Prepare$\rangle$ message will be accepted by a backup p_j provided that:

- It verifies the MAC;
- The message digest is correct;
- It is in view v;
- $seq = seq_l + 1$, where seq_l is the sequence number of its last accepted request;
- It has not accepted a $\langle$Prepare$\rangle$ message with the same sequence number in the same view but contains a different request.

1. Using MACs instead of digital signatures usually makes protocols much faster. In Aardvark [11], on a 2.0GHz Pentium-M, openssl 0.9.8g can compute over 500,000 MACs per second for 64 byte messages, but it can only verify 6455 1024-bit RSA signatures per second or produce 309 1024-bit RSA signatures per second.

If a backup p_j accepts the $\langle$Prepare$\rangle$ message, it speculatively executes the operation and sends a reply message $\langle \text{Reply}, v, t, seq, \delta_{seq}, c\rangle$ to c and also a commit message $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c\rangle$ to all replicas, where δ_{seq} contains the speculative execution history.

In order to verify the correctness of the speculatively executed request, a replica collects $2f+1$ matching $\langle$Commit$\rangle$ messages from other replicas to complete a request. As shown in Algorithm 2, a replica collects matching $\langle$Commit$\rangle$ messages. If a replica receives $f + 1$ matching $\langle$Commit$\rangle$ messages from different replicas but has not accepted any $\langle$Prepare$\rangle$ message, it also speculatively executes the operation, sends a $\langle$Commit$\rangle$ message to all replicas, and sends a reply to the corresponding client. When the replica collects $2f$ matching messages, it puts the corresponding request in its speculative execution history and completes the request. However, it is possible that a replica receives $f + 1$ matching $\langle$Commit$\rangle$ messages from other replicas that are conflicting with its accepted $\langle$Prepare$\rangle$ message. Under such circumstance, the replica can simply send a $\langle$View-Change$\rangle$ message to all replicas. If a replica votes for view change, it stops receiving any messages except the $\langle$New-View$\rangle$ and the checkpoint messages. See Section 4.3 for the detail of view change subprotocol.

The exchange of $\langle$Commit$\rangle$ messages is to ensure that if at least $f + 1$ correct replicas speculatively execute a request, all the correct replicas learn the result. If any other correct replicas receive inconsistent messages, the primary must be faulty and the replicas stop receiving messages until view change occurs.

A client sets a timeout for each request. As shown in Algorithm 3, a client collects matching $\langle$Reply$\rangle$ messages to its request. If it gathers $2f+1$ matching speculative replies from different replicas before the timeout expires, it completes the request. If a client receives fewer than $f + 1$ matching replies before the timeout expires, it retransmits the requests. Otherwise, when client receives between $f+1$ to $2f+1$ matching replies before timeout expires, it facilitates the progress by sending a $\langle \text{PANIC}, D(m), t, c\rangle_c$ message to all replicas. If a replica receives a $\langle$PANIC$\rangle$ message, it forwards the message to all replicas. If a replica does not receive any $\langle$PANIC$\rangle$ message from the client but receives a $\langle$PANIC$\rangle$ message from other replicas, it forwards the $\langle$PANIC$\rangle$ message to all replicas. A $\langle$PANIC$\rangle$ message is valid if a replica has speculatively executed m. If a replica accepts a $\langle$PANIC$\rangle$ message, it stops receiving any messages except the view change and checkpoint messages.

There are two goals for replicas when forwarding $\langle$PANIC$\rangle$ messages. One is to prevent the checkpoint protocol from occurring too frequently, which happens when all the correct replicas receive the $\langle$PANIC$\rangle$ message before the checkpoint protocol is triggered. Another is to prevent the clients from attacking the

system by sending ⟨PANIC⟩ messages to a portion of the replicas. If a faulty client sends a ⟨PANIC⟩ message to a correct backup, the replica will stop receiving any messages while other replicas still continue the agreement protocol. This forwarding mechanism ensures that if at least one correct replica receives the ⟨PANIC⟩ message, all the replicas receive the ⟨PANIC⟩ message and enter the checkpoint protocol.

Algorithm 1 Primary

Initialization:
A {All replicas}
$seq \leftarrow 0$ {Sequence number}
$\mathcal{W}$ {Set of ⟨PANIC⟩ messages}

on event $\langle \text{Request}, o, t, c \rangle_c$
  $seq \leftarrow seq + 1$
  send $\langle \text{Prepare}, v, seq, D(m), m, c \rangle$ to A
  send $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c

on event $\langle \text{PANIC}, D(m), t, c \rangle_c$ from c
  send $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A

on event $\langle \text{PANIC}, D(m), t, c \rangle_c$ from A
  if $match(\mathcal{W}_c)$ **then**
    $\mathcal{W}_c.add$ {Add matching ⟨PANIC⟩ message}
  if $\mathcal{W}_c.size = 2f + 1$ **then**
    Initialize checkpoint protocol

Algorithm 2 Backup

Initialization:
A {All replicas}
$seq_i \leftarrow 0$ {Sequence number}
$\mathcal{U}$ {Set of ⟨Commit⟩ messages}
$panic \leftarrow F$ {If true, enter checkpoint protocol}

on event $\langle \text{Request}, o, t, c \rangle_c$
  send $\langle \text{Request}, o, t, c \rangle_c$ to the primary

on event $\langle \text{Prepare}, v, seq, D(m), m, c \rangle$
  if $seq = seq_i + 1$ **then**
    $seq_i \leftarrow seq$
    send $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$ to A
    send $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c

on event $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$
  if $match(\mathcal{U}_{seq})$ **then**
    $\mathcal{U}_{seq}.add$ {Add matching ⟨Commit⟩ message}
  if $\mathcal{U}_{seq}.size = f + 1$ **and** $seq = seq_i + 1$ **then**
    $seq_i \leftarrow seq$ {Accept the message}
    send $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$ to A
    send $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c
  if $\mathcal{U}_{seq}.size = 2f$ **and** $seq = seq_i$ **then**
    $complete(\mathcal{U}_{seq})$ {Complete the request}

on event $\langle \text{PANIC}, D(m), t, c \rangle_c$
  if $panic = F$ **then**
    send $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A
    $panic \leftarrow T$ {Enter checkpoint protocol}

Algorithm 3 Client

Initialization:
A {All replicas}
$\mathcal{V}$ {Set of ⟨Reply⟩ messages}
send $\langle \text{Request}, o, t, c \rangle_c$ to A
$start(\Delta)$ {Start a timer}

on event $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$
  if $match(\mathcal{V}_{seq})$ **then**
    $\mathcal{V}_{seq}.add$ {Add matching ⟨Reply⟩ message}
  if $\mathcal{V}_{seq}.size = 2f + 1$ **then**
    $cancel(\Delta)$ {Complete the request}

on event $timeout(\Delta)$
  if $\mathcal{V}_{seq}.size < f + 1$ **then**
    retransmit $\langle \text{Request}, o, t, c \rangle_c$ to A
  else
    send $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A

The primary initializes the checkpoint subprotocol if it receives the ⟨PANIC⟩ message from the client or $2f + 1$ ⟨PANIC⟩ messages from other replicas. The correctness of the protocol is therefore guaranteed by the three-phase checkpoint subprotocol.

The panic mechanism facilitates progress when the primary is faulty. Specifically, in a partial synchrony model where the value of a client's timeout is properly set up, if a correct client does not receive sufficient matching replies before timer expires, the primary either sends inconsistent ⟨Prepare⟩ messages to the replicas or fails to send consistent messages to the replicas. In this case, instead of using the traditional approach where replicas detect the faulty primary themselves by waiting for longer period of time, the client can directly trigger the checkpoint protocol in order to verify the correctness of the primary. See Section 4.2 for details of the checkpoint subprotocol.

*h*BFT guarantees correctness while using only two phases. If the client has received $2f + 1$ matching replies, at least $f+1$ correct replicas receive consistent order from the primary. Therefore, all correct replicas receive at least $f+1$ matching ⟨Commit⟩ messages. If those replicas do not receive the ⟨Prepare⟩ message, they will execute the request. Otherwise, if they detect the inconsistency, they stop receiving any messages until the current primary is replaced or the checkpoint subprotocol is triggered. In the latter case, the inconsistency will be reflected and fixed in the checkpoint subprotocol.

4.2 Checkpoint

We use a three-phase PBFT-like checkpoint protocol. The reasons are three-fold. First, the agreement protocol uses speculative execution and replicas may be temporarily out of order. The three-phase checkpoint protocols resolve the inconsistencies. Second, if a correct client triggers the checkpoint protocol through the panic mechanism, the checkpoint protocol resolves

the inconsistencies immediately. Third, the checkpoint protocol detects the behavior of the faulty clients if they intentionally trigger the checkpoint protocol.

The checkpoint protocol works as follows. Only the primary can initialize the checkpoint subprotocol, which is generated under either of the two conditions:

- the primary executes a certain number of requests
- the primary receives $2f + 1$ forwarded $\langle$PANIC$\rangle$ messages from other replicas.

In the latter condition, as mentioned in Section 4.1, when a replica receives a valid $\langle$PANIC$\rangle$ message, it forwards to all replicas. The goal is to ensure that all replicas receive the $\langle$PANIC$\rangle$ message and also to prevent faulty clients from sending a $\langle$PANIC$\rangle$ message only to the backups, thereby making sure replicas will not erroneously suspect the primary due to the faulty clients.

The three-phase checkpoint subprotocol works as follows: the current primary p_i sends a $\langle$Checkpoint-I$, seq, D(M)\rangle$ to all replicas, where seq is the sequence number of last executed operation, $D(M)$ is the message digest of speculative execution history M. Upon receiving a well-formatted $\langle$Checkpoint-I$\rangle$ message, a replica sends a $\langle$Checkpoint-II$, seq, D(M)\rangle$ to all replicas. If the digest and execution history do not match its local log, the replica sends a $\langle$View-Change$\rangle$ message directly to all replicas and stops receiving any messages other than the $\langle$New-View$\rangle$ message.

A number of $2f + 1$ matching $\langle$Checkpoint-II$\rangle$ messages from different replicas form a certificate, denoted by $\mathcal{CER}_1(M, v)$. Any replica p_j that has the certificate sends a $\langle$Checkpoint-III$, seq, D(M)\rangle_j$ to all replicas. Similarly, $2f + 1$ $\langle$Checkpoint-III$\rangle$ messages form a certificate, denoted by $\mathcal{CER}_2(M, v)$. After collecting $\mathcal{CER}_2(M, v)$, the checkpoint becomes stable. All the previous checkpoint messages, $\langle$Prepare$\rangle$, $\langle$Commit$\rangle$, $\langle$Request$, o, t, c\rangle_c$, and $\langle$Reply$\rangle$ messages with smaller sequence number than the checkpoint are discarded.

If a view change occurs in the checkpoint subprotocol, as described in Section 4.3, the new primary initializes a checkpoint immediately after the $\langle$New-View$\rangle$ message. The same three-phase checkpoint subprotocol continues until one checkpoint is completed and the system stabilizes.

4.3 View Changes

The view change subprotocol elects a new primary. By default, the primary has id $p = v \bmod n$, where n is the total number of replicas and v is the current view number. View changes may take place in the checkpoint protocol or the agreement protocol. In both cases, the new primary reorders requests using a $\langle$New-View$\rangle$ message and then initializes a checkpoint immediately. The checkpoint subprotocol continues until one checkpoint is committed.

A $\langle$View-Change$, v + 1, \mathcal{P}, \mathcal{Q}, \mathcal{R}\rangle_i$ message will be sent by a replica if any of the following conditions are true, where $\mathcal{P}$ contains the execution history M from $\mathcal{CER}_1(M, v)$ the replica collected in previous view v, $\mathcal{Q}$ denotes the execution history from the accepted $\langle$Checkpoint-I$\rangle$ message, and $\mathcal{R}$ denotes the speculatively executed requests with sequence numbers greater than its last accepted checkpoint:

- It starts a timer for the first request in the queue. The request is not executed before the timer expires;
- It starts a timer after collecting $f + 1$ $\langle$PANIC$\rangle$ messages. It has not received any checkpoint messages before the timer expires;
- It starts a timer after it executes certain number of requests. It has not received any checkpoint messages before the timer expires;
- It receives $f + 1$ valid $\langle$View-Change$\rangle$ messages from other replicas.

Timers with different values are set for each case and are reset periodically.

When the new primary p_j receives $2f$ $\langle$View-Change$\rangle$ messages, it constructs a $\langle$New-View$\rangle$ message to order all the speculatively executed requests. The system then moves to a new view. The principle is that any request committed by the clients must be committed by all correct replicas. The new primary picks up an execution history M from $\mathcal{P}$ and a set of requests from the $\mathcal{R}$ of checkpoint messages. To select a speculative execution history M, there are two rules.

A If some correct replica has committed on one checkpoint that contains execution history M, M must be selected, provided that:
  A1. At least $2f + 1$ replicas have $\mathcal{CER}_1(M, v)$.
  A2. At least $f + 1$ replicas have accepted $\langle$Checkpoint-I$\rangle$ in view $v' > v$.

B If at least $2f + 1$ replicas have empty $\mathcal{P}$ components, then the new primary selects its last stable checkpoint.

Similarly, for each sequence number greater than the execution history M and smaller than the largest sequence number in $\mathcal{R}$ of checkpoint messages, the primary assigns a request according to $\mathcal{R}$. A request m is chosen if at least $f + 1$ replicas include it in $\mathcal{R}$ of their checkpoint messages. Otherwise, NULL is chosen. We claim that it is impossible for $f + 1$ replicas to include one request m, and another $f + 1$ replicas include m' with the same sequence number. Namely, if $f + 1$ replicas include a request m, at least one correct replica receives $2f+1$ $\langle$Commit$\rangle$ messages. Similarly, at least one correct replica receives $2f + 1$ commit messages with request m'. The two quorums intersect in at least one correct replica. The correct replica must have sent both $\langle$Commit$\rangle$ message with m and $\langle$Commit$\rangle$ message with m', a contradiction.

The execution history M and the set of requests form M', which is composed of requests with se-

quence numbers between the last stable checkpoint and the sequence number that has been used by at least one correct replica. The new primary then sends a $\langle \text{New-View}, v+1, \mathcal{V}, \mathcal{X}, M' \rangle_j$ message to all replicas, where $\mathcal{V}$ contains $f+1$ valid $\langle$View-Change$\rangle$ messages, $\mathcal{X}$ contains the selected checkpoint. The replicas then run the checkpoint subprotocol using M'. The checkpoint subprotocol continues until one checkpoint is committed.

4.4 Client Suspicion

Faulty clients may render the system unusable, especially for protocols that move some critical jobs to the clients. In *h*BFT, unlimited numbers of faulty clients can be detected. We focus on the "legal" but problematic messages a faulty client can craft to slow down the performance or cause incorrectness. To be specific, a faulty client can do the following:

- It sends inconsistent requests to different replicas. The primary may not be able to order "every" request before the timeout expires. In this case, a correct primary may be removed.
- It intentionally sends $\langle$PANIC$\rangle$ messages while there is no contention. Unnecessary checkpoint subprotocol will be triggered, which slows down the performance. However, if the client frequently triggers "valid" checkpoint operations, the overall throughput decreases too.
- It does not send $\langle$PANIC$\rangle$ messages if it receives divergent replies, leaving replicas temporarily inconsistent.

The client suspicion subprotocol in *h*BFT focuses on the first two. If the third one occurs, the checkpoint subprotocol can be triggered by the next correct client if it detects the divergence of replies or by the primary when replicas execute certain number of requests.

To solve the first problem, we ask clients to multicast the request to the replicas and every replica forwards the request to the primary. The primary orders a request if it receives the request or if it receives $f+1$ matching requests forwarded by backups. If a replica p_i receives a $\langle$Prepare$\rangle$ message with a request that is not in its queue, it still executes the operation. Nevertheless, such faulty behavior of clients will be identified as suspicious, and if the number of suspicious incidents from the same client exceeds certain threshold, p_i will send a $\langle \text{Suspect}, c \rangle_i$ message to all replicas.

Another reason clients send their requests to all replicas is that there are many drawbacks when clients send requests only to the primary.[2] For instance, a faulty primary can delay any request, regardless of whether the primary receives the request from the client or other replicas. This would cause all clients to multicast their requests to all replicas. In other words, a faulty primary makes all clients experience long latency without being noticed. A faulty primary can also perform a performance attack such as timeout manipulation, as discussed in other work [2, 11, 29]. Furthermore, it is also difficult to make clients keep track of the primary. If the client sends its request to a faulty backup, the faulty backup can also ignore this request, although it is supposed to forward the request to the primary. In many existing protocols, all of these problems typically mean that the primary task for establishing correctness is the process of detecting faulty replicas.

For the second problem where a faulty client intentionally sends a $\langle$PANIC$\rangle$ message to the replicas to trigger the checkpoint subprotocol, the protocol naturally detects the faulty behavior. Intuitively, if the request is committed in both agreement protocol and checkpoint protocol without view change, the client can be suspected. Nevertheless, a correct client might be suspected as well. For instance, the following two cases are indistinguishable.

(1) The replicas are correct and reach an agreement in the agreement protocol. When they receive the $\langle$PANIC$\rangle$ message from a faulty client, the request is committed in the checkpoint protocol without view change and the client is suspected.
(2) The primary is faulty and the client is correct. The primary sends the request to $f+1$ correct replicas and another fake request to the remaining f correct replicas. The f correct replicas will not execute th request. When the replicas receive $\langle$PANIC$\rangle$ message and starts checkpoint protocol, the f faulty replicas collude and make the request committed in the checkpoint protocol. Although the f correct replicas learn the result and remain consistent, the correct client will be suspected.

To distinguish the above two cases, we modify the agreement protocol by simply replacing the MACs of $\langle$Prepare$\rangle$ messages with digital signatures, which is called *Almost-MAC-agreement*. When a replica sends a $\langle$Commit$\rangle$ message, it appends the $\langle$Prepare$\rangle$ message. If a client does not receive valid $\langle$Prepare$\rangle$ message from the primary but receives from other replicas, it still executes the requests, sends $\langle$Commit$\rangle$ messages to other replicas, and sends a $\langle$Reply$\rangle$ to the client. Otherwise, if a replica receives two valid and conflicting $\langle$Prepare$\rangle$ messages, it directly sends inconsistent messages to all replicas and votes for view change. As proven in Claim 2, the protocol guaranteed that correct clients will not be removed. This optimization can also solves the problem as discussed in Section 5.1.

The modification of agreement protocol results in $2 + \frac{1(sig)}{b}$ cryptographic operations for the primary.

2. In some Byzantine agreement protocols, clients send requests only to their known primary. If a backup receives the request, it forwards the request to the primary, expecting the request to be executed. The client sets a timeout for each request it has. If it does not receive sufficient matching responses before timeout expires, it retransmits the request to all replicas.

To reduce the overall cryptographic operations, *h*BFT switches between the agreement protocol and Almost-MAC-agreement when executing a certain number of requests.

The client will only be suspected when replicas are running Almost-MAC-agreement. In addition, the client must be suspected by $2f + 1$ replicas to be removed. If the number of such incidents exceeds certain threshold, replicas will suspect the client and send a ⟨Suspect⟩ message to all replicas. Similarly to the view change subprotocol, if a replica receives $f+1$ ⟨Suspect⟩ messages, it generates a ⟨Suspect⟩ message and sends to the replicas. If a replica receives $2f + 1$ ⟨Suspect⟩ messages, indicating that at least one correct replica suspects the client, the client can be prevented from accessing the system in the future.

Worst Case Scenario We would like to analyze the worst case where a correct client can be suspected, mainly due to the network failure. It happens if any of the following is true:

(1) The request from client fails to reach $f + 1$ correct backups before the backups receive the ⟨Prepare⟩ message. In this case, since the $f + 1$ correct backups do not receive the request in the ⟨Prepare⟩ message, they will suspect the client.
(2) ⟨Reply⟩ messages from correct replicas fail to reach the client before the timeout expires. Since the client does not receive $2f + 1$ matching replies before the timeout expires, the client sends ⟨PANIC⟩ messages while there is no contention.

The latter condition may occur due to an inappropriate value of the timeout regarding the network condition or due to the attack by the primary. For instance, a faulty primary can intentionally delay ⟨Prepare⟩ messages for some correct replicas, causing correct clients to send a ⟨PANIC⟩ message even though replicas are "consistent." However, if the value of the timeout is appropriately set up using Almost-MAC-agreement, as proven in Claim 2, correct clients will not be removed. To set up an appropriate value, the clients adjust the values of the timeout during retransmission. Namely, when the client retransmits the request, it doubles the timeout and starts again. In this case, the value of the timeout will eventually be large enough for the client to receive ⟨Reply⟩ messages.

4.5 Correctness

In this section, we sketch proofs for the safety and liveness properties of *h*BFT under optimal resilience. For simplicity, we assume there are $3f + 1$ replicas.

4.5.1 Safety

Theorem 1 (Safety): If requests m and m' are committed at two correct replicas p_i and p_j, m is committed before m' at p_i if and only if m is committed before m' at p_j.

Proof: The proof proceeds as follows. We first prove the correctness of checkpoint subprotocol, which follows the correctness of PBFT, as shown in *Claim 1*. We then show the proof of the theorem based on the claim.

Claim 1 (Safety of Checkpoint): The checkpoint subprotocol guarantees the safety property.

Proof: We now prove that if checkpoints M and M' are committed at two correct replicas p_i and p_j in checkpoint subprotocol, regardless of being in the same view or across views, $M = M'$.

(Within a view) If p_i and p_j commit both in view v, then p_i has collected $\mathcal{CER}_2(M, v)$, which indicates that at least $f+1$ correct replicas have sent ⟨Checkpoint-III⟩ for M. Similarly, p_j has $\mathcal{CER}_2(M', v)$, which indicates that at least $f+1$ correct replicas send ⟨Checkpoint-III⟩ for M'. Then excluding f faulty replicas, if M and M' are different, at least one correct replica has sent two conflicting messages for M and M', which contradicts with our assumption. Therefore, $M = M'$.

(Across views) If M is committed at p_i in view v and M' is committed at p_j in view $v' > v$, $M = M'$. If M' is committed in view v', then either condition A or B must be true in the construction of the ⟨New-View⟩ message in view v' (see Section 4.3). However, if M is committed at p_j in view v, p_j has $\mathcal{CER}_2(M, v)$, which indicates that at least $f + 1$ correct replicas have $\mathcal{CER}_1(M, v)$ and M in the $\mathcal{P}$ component. Therefore, condition B cannot be true. For condition A, M' is committed at p_j in view v' if both A1 and A2 are true. A2 can be true if a faulty replica sends a ⟨View-Change⟩ message that includes $\langle M', D(M'), v_1\rangle$, where $v < v_1 \leq v'$. However, condition A1 requires that at least $f + 1$ correct replicas have $\mathcal{CER}_1(M', v')$. Since at least $f+1$ correct replicas have $\mathcal{CER}_1(M, v)$, they will not accept M' in any later views. At least one correct replica sends conflicting messages, a contradiction. Therefore, we have $M = M'$. □

To prove Theorem 1, we first show that if two requests m and m' are committed at correct replicas p_i and p_j, m equals m'. Then we show that if m_1 is committed before m_2 at p_i, m_1 is committed before m_2 at p_j. The former part is shown across views and within the same view.

(Within a view) There are three cases: the two requests are committed in agreement subprotocol, two requests are committed in checkpoint subprotocol, one of them is committed in the agreement subprotocol and the other one is committed in the checkpoint subprotocol. In the first case, if m is committed at p_i, p_i receives $2f + 1$ ⟨Commit⟩ messages if the request is committed in agreement protocol. On the other hand, if m' is committed at p_j, p_j receives $2f + 1$ ⟨Commit⟩ messages. The two quorums intersect in at least one correct replica. At least one correct replica sends inconsistent messages, a contradiction. Therefore, m equals m'. The second case is proved in

Claim 1. In the third case, if m is committed at p_i, p_i receives $2f + 1$ ⟨Commit⟩ messages if the request is committed in the agreement protocol. On the other hand, if m' is committed at p_j in checkpoint protocol, at least $2f + 1$ replicas have certificate with m' in their execution history. The two quorums of $2f + 1$ replicas intersect in at least one correct replica, who sends a ⟨Commit⟩ message with m in the agreement protocol and includes m' in its execution history in the checkpoint protocol, a contradiction. To summarize, we have m equals m' if they are committed in the same view.

(Across views) If m is committed at replica p_j, $2f + 1$ replicas send ⟨Commit⟩ messages. At least $f+1$ correct replicas accept m, which will be included in their ⟨View-Change⟩ messages. On every view change, the new primary initializes a checkpoint subprotocol to make the same order of requests committed at all the correct replicas in the ⟨New-View⟩ message. The correctness follows from Claim 1.

Then we show that if m_1 is committed before m_2 at p_i, m_1 is committed before m_2 at p_j. If a request is committed at a correct replica, $2f + 1$ replicas send ⟨Commit⟩ messages. Since two quorums of $2f + 1$ replicas intersect in at least one correct replica p_i, m_1 is committed with sequence number smaller than m_2. According to the former proof, if m_1 and m_2 are committed at p_j, they are committed with the same sequence numbers.

By combining all the above, safety is proven. □

4.6 Liveness

Theorem 2 (Liveness): Correct clients eventually receive replies to their requests.

Proof: It is trivial to show that if the primary is correct, clients receive replies to their requests. In the following, we first show that correct clients will not be removed. We then prove that faulty replicas and faulty clients cannot impede progress by removing a correct primary.

Claim 2 (Correct Client Condition): If the values of the timeouts are appropriately set up, correct clients will not be removed if they trigger a checkpoint.

Proof: If a correct client receives between $f + 1$ to $2f + 1$ matching replies for a request m, it triggers the checkpoint subprotocol. To remove a correct client, m must be executed by $f + 1$ replicas in Almost-MAC-agreement protocol and committed in the checkpoint subprotocol without view changes. Among the $f + 1$ replicas that accept ⟨Prepare⟩ message in the agreement protocol, at least one is correct. If it receives a ⟨Prepare⟩ message, it appends to ⟨Commit⟩ message and sends to all replicas. If at least one correct replica receives a valid and conflicting ⟨Prepare⟩ message from the primary, it will send inconsistent messages and eventually all the correct replicas vote for view change, a contradiction that view change does not occur. Therefore, no correct replica receives a different ⟨Prepare⟩ message. In addition, if a correct replica does not receive a valid ⟨Prepare⟩ message from the primary and receives a valid ⟨Prepare⟩ message appended to the ⟨Commit⟩ message, it will accept the ⟨Prepare⟩ message and sends ⟨Reply⟩ message to the client. In this case, the client receives $2f + 1$ matching replies, a contradiction with the assumption that the client is correct. Therefore, correct clients will not be removed by the client suspicion protocol. □

Claim 3 (Faulty Replica Condition): Faulty replicas cannot impede progress by causing view changes.

Proof: To begin, we show that faulty replicas cannot cause a view change by sending ⟨View-Change⟩ messages. At least $f + 1$ ⟨View-Change⟩ messages are sufficient to cause a view change. Thus, even if all faulty replicas vote for view change, they cannot cause a view change. A faulty primary *can* cause a view change. However, the primary cannot be faulty for more than f consecutive views.

In addition, no ⟨View-Change⟩ message makes a correct primary incapable of generating a ⟨New-View⟩ message. A correct primary is able to pick up a stable checkpoint. Since at least $f + 1$ correct replicas have $\mathcal{CER}_2$ for a checkpoint, the new primary is able to pick it up. In addition, the new primary is able to pick up a sequence of requests based on condition A or B. Either some correct replica(s) commits on a checkpoint or no correct replica does. Condition A1 can be verified because non-faulty replicas will not commit on two different checkpoints. Condition A2 is satisfied if at least one correct replica accepts a ⟨Checkpoint-I⟩ message for the same checkpoint and it votes for the authenticity of the checkpoint. Therefore, the checkpoint can be selected since it is authentic. Similarly, a set of executed requests can be selected based on $\mathcal{R}$ in a view change. Namely, if the client completes a request, the request must be accepted by at least $2f + 1$ replicas. Among them, at least $f+1$ replicas are correct. If other replicas receive inconsistent ⟨Prepare⟩ messages and $f + 1$ ⟨Commit⟩ messages, they will abort. Therefore, it is not possible that a set of $f + 1$ replicas include one request and another set of $f+1$ replicas include another request. In conclusion, the new primary is able to select a ⟨New-View⟩ message. □

Claim 4 (Faulty Client Condition 2): A faulty client cannot impede progress by causing view changes.

Proof: If a faulty client intentionally triggers the checkpoint subprotocol while replicas are consistent, requests committed in agreement subprotocol will be committed in checkpoint subprotocol. View changes will not occur. Since such faulty behavior of clients will be detected, the client will be removed. □

To summarize, according to Claim 2, correct replicas will not be removed so their requests can be handled. Faulty backups or faulty clients can not cause view changes as proved in Claim 3 and Claim 4 separately.

Since the primary cannot be faulty for more than f continuous views, correct clients eventually receive replies to their requests. □

5 Discussion

5.1 Timeouts

Existing protocols rely on different timeouts to guarantee liveness. As discussed in Section 4.4, the values of timeouts are key to avoid some uncivil attacks. Since we assume the partial synchrony model, it is reasonable to set up timeouts according to the round-trip time such as the technique used in Prime [2]. However, in several corner cases, either inappropriate values of timeouts or network congestion can make a correct replica suspect or remove a correct primary.

*h*BFT employs a client suspicion subprotocol that is used to detect faulty clients. A faulty primary can play tricks on timeouts to remove correct clients. For instance, the primary can send a ⟨Prepare⟩ message to f correct replicas and delay the ⟨Prepare⟩ message to $f+1$ correct replicas until the very end of timeout of the client. The $f+1$ correct replicas receive the ⟨Prepare⟩ message and execute the request but they do not reply to the clients "on time." Since the client does not receive enough number of replies before the timeout expires, it sends a ⟨PANIC⟩ message. However, all replicas are "consistent" since the primary still sends out consistent ⟨Prepare⟩ messages. Correct clients will be suspected.

We solve this problem by using Almost-MAC-agreement protocol as discussed in Section 4.4. The optimization allows all replicas to execute the request on time if at least one correct replica receives a valid ⟨Prepare⟩ message, which prevents a faulty primary from framing the clients.

5.2 Speculation

Speculation reduces the cost and simplifies the design of Byzantine agreement protocols, which works well especially for systems with highly concurrent requests. Speculation has been used by fault-free systems and by systems that tolerate crash failures. Therefore, *h*BFT also works well in adaptively tolerating crash failures to Byzantine failures. *h*BFT uses speculation because replicas are always consistent for both fault-free and normal cases where the primary is correct. Every request takes three communication steps to complete, and is the theoretical lower bound for agreement-based protocols.

Speculation does not work well for systems that have high computationally intensive tasks or systems that have a high attack rate. The former problem can be handled by separating execution from agreement [32]. The latter problem decreases the performance either with or without recovery. For instance, faulty clients can simply trigger the three-phase checkpoint subprotocol on every request, which gives *h*BFT similar performance to PBFT before the faulty clients are evacuated. The advantage of *h*BFT indicates that the three-phase checkpoint subprotocol is rarely triggered. Therefore, *h*BFT improves the performance in fault-free and normal cases but achieves comparable performance to PBFT in the worst case.

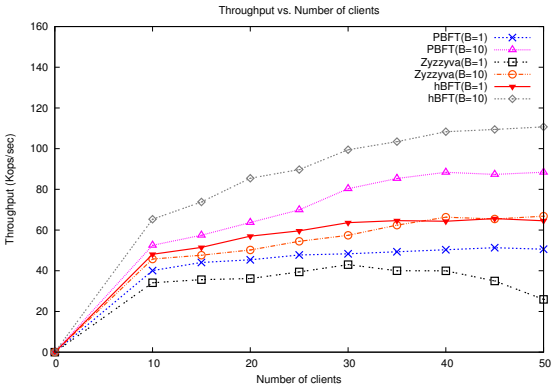


Fig. 5. Throughput for the $0/0$ benchmark as the number of clients varies for systems to tolerate $f = 1$ faults.

6 Evaluation

We evaluate the system on Emulab [31] utilizing up to 45 $pc3000$ machines connected through a 100Mbps switched LAN. Each machines is equipped with a 2GHz, 64-bit Xeon processor with 2GB of RAM. 64-bit Ubuntu 10 is installed on every machine, running Linux kernel 2.6.32. We use RSA-FDH [4] for our digital signature scheme, and HMAC-MD5 [5, 6] for the MAC algorithm.

We compare our work with Castro et al.'s implementation of PBFT [7] as well as Kotla et al.'s implementation of Zyzzyva [21]. All the experiments are carried out in normal cases, where a backup is faulty. Four micro-benchmarks are used in the evaluation, also developed by Castro et-al. An x/y benchmark refers to an xkB request from clients and an ykB reply from the replicas.

6.1 Throughput

Fig. 5 compares throughput achieved for the $0/0$ benchmark in normal cases between PBFT, Zyzzyva and *h*BFT where B is the size of the batch. Fig. 6 presents the performance for the four benchmarks where $B = 1$ for all benchmarks. All the experiments are tested in the configuration of $f = 1$.

As the number of clients increases, Zyzzyva performs even worse than PBFT. As indicated in Section 1.1, without batching $(B = 1, f = 1)$, bottleneck server of Zyzzyva $(4 + 5f + \frac{3f}{b})$ performs 1.2 times more MAC operations than PBFT $(2 + \frac{8f}{b})$ and 2.4 times more MAC operations than *h*BFT $(2 + \frac{3f}{b})$. With batching $(B = 10, f = 1)$, Zyzzyva performs 3.3 times

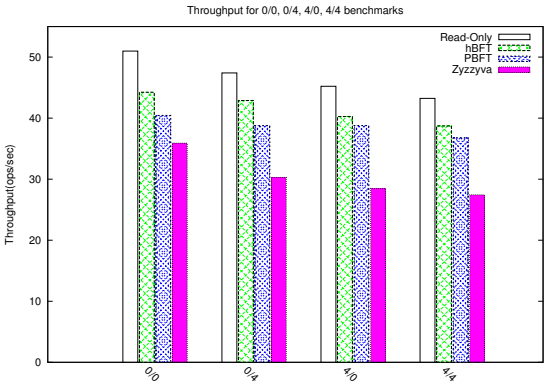


Fig. 6. Throughput for $0/0, 0/4, 4/0$ and $4/4$ benchmarks for systems to tolerate $f = 1$ faults.

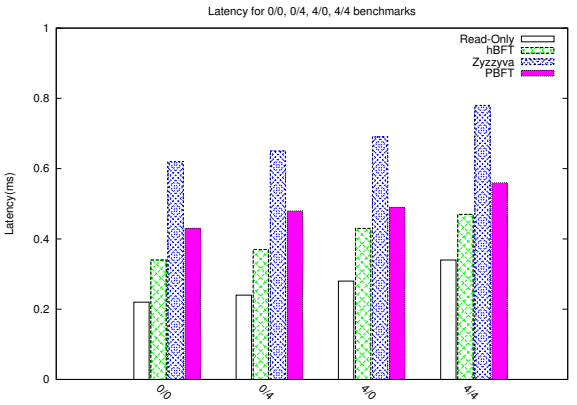


Fig. 7. Latency for $0/0, 0/4, 4/0$ and $4/4$ benchmarks for systems to tolerate $f = 1$ faults without contention.

more MAC operations than PBFT and 4.0 times more MAC operations than *h*BFT.

The simulation validates the theoretical results. As shown in Fig. 5, without batching, *h*BFT achieves more than 40% higher throughput than PBFT and 20% higher throughput than Zyzzyva. With batching, the peak throughput of *h*BFT is 2 times better than that of Zyzzyva, and 40% higher than that of PBFT. The difference is due to the cryptographic overhead of each protocol.

Additionally, *h*BFT outperforms both Zyzzyva and PBFT under high concurrency. As the number of clients grows, all three protocols achieve better performance with batching than without. When the number of clients exceeds 40, throughput of Zyzzyva degrades obviously. All other cases remain stable when the number of clients exceeds 30. When the number of clients is fewer than 30, *h*BFT with batching has an outstanding growth. Other than that, throughput of PBFT with batching also grows faster compared with all the left cases. The reply message cannot be batched and replicas need to reply to every client, which explains the result why Zyzzyva achieves the lowest throughput in normal cases.

Fig. 6 presents the throughput of protocols without batching with 10 clients. For all the benchmarks, *h*BFT achieves higher throughput as well. All three protocols achieve the best throughput for $0/0$ benchmark and the worst for $4/4$ benchmark. Zyzzyva and *h*BFT perform worse for $0/4$ and $4/4$ benchmarks than $4/0$ benchmark. PBFT achieves almost the same throughput for $0/4$ and $4/0$ benchmarks. This implies that the size of reply messages has more effect for speculation-based protocols.The outstanding performance of read-only requests is due to the read-only optimization, where replicas send reply directly to the clients without running agreement protocol.

To summarize this section, *h*BFT outperforms both Zyzzyva and PBFT in normal cases. Since PBFT achieves almost the same throughput for $0/4$ and $4/0$ benchmarks and it achieves higher throughput with

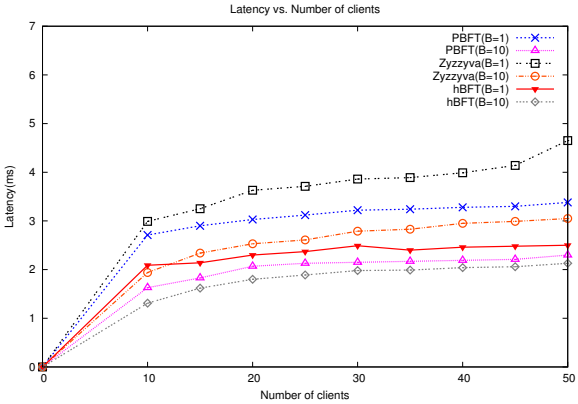


Fig. 8. Latency for the $0/0$ benchmark as the number of clients varies for systems to tolerate $f = 1$ faults.

batching, it works well for systems that have more computationally consuming tasks. Comparably, *h*BFT and Zyzzyva work well for systems that have highly concurrent but lightweight requests.

6.2 Latency

The latency depends on both cryptographic overhead and one-way latencies. Cryptographic overhead controls the latency of processing one message and the number of one-way latencies controls the number of phases that the agreement protocol goes through. In terms of critical paths, PBFT has four if replicas send reply to the clients after prepare phase. *h*BFT has only three, which is the theoretical lower bound of agreement protocols under high concurrency. Even though the checkpoint subprotocol takes three phases, it will not decrease the overall performance significantly since the checkpoint subprotocol is triggered rarely. Zyzzyva takes three in fault-free cases and five in normal cases.

Additionally, the performance of all protocols is also related to the frequency of checkpoint subprotocol as well. It has a direct impact on *h*BFT due to the reason

that checkpoint subprotocol of *h*BFT is more expensive than the other two. By default, we assume that a checkpoint subprotocol starts every 128 requests. *h*BFT outperforms the other two under this setting. If we make checkpoint subprotocol more rarely, it can be expected that *h*BFT will achieve even better performance and vice versa.

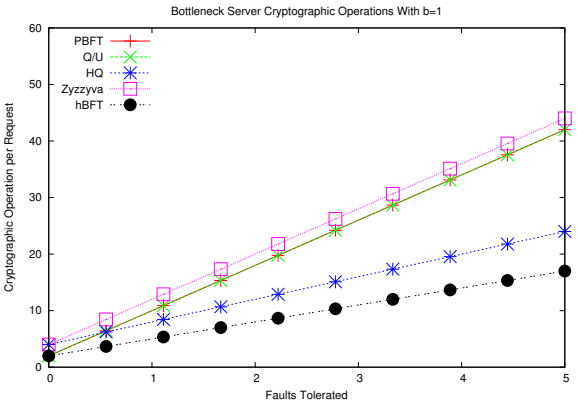
We assess the latency without contention when there is only 1 client. The performance for all four benchmarks are similar, as shown in Fig. 7. All three protocols have the lowest latency for the $0/0$ benchmark and the highest for the $4/4$ benchmark. PBFT achieves almost the same latency for both $4/0$ and $0/4$ benchmarks. *h*BFT and Zyzzyva achieve lower latency for the $4/0$ benchmark than the $0/4$ benchmark.

As shown in Fig. 8, we also evaluate latency as the number of clients grows. We observe that without batching, *h*BFT achieves an average of 30% lower latency than PBFT and 40% lower latency than Zyzzyva. With batching, *h*BFT achieves an average of 15% lower latency than PBFT and 35% lower latency than Zyzzyva. When the number of clients increases, the latency of all the protocols increase gradually, which shows that all three protocols work well under high concurrency. The latency of Zyzzyva grows faster than the other two.

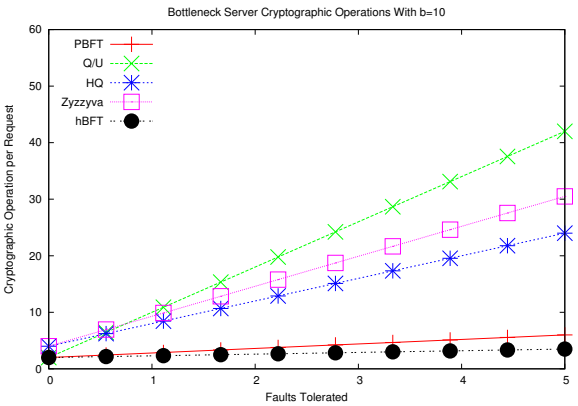
6.3 Fault Scalability

We also examine performance when the number of replicas increases. As shown in Fig. 1, the throughput is related to f. We view the primary as the bottleneck server not only because of the number of MAC operations in the agreement, but also because of other effort such as processing requests. For PBFT and *h*BFT, the backups do not perform many fewer cryptographic operations than the primary. In comparison, backups in Zyzzyva perform many fewer cryptographic operations than the primary, which can be viewed as an advantage over the other two. However, this does not have a direct positive effect on the throughput and latency since the primary performs more cryptographic operations. As f increases, the performance for all three protocols will decrease due to the cryptographic overhead, especially without batching.

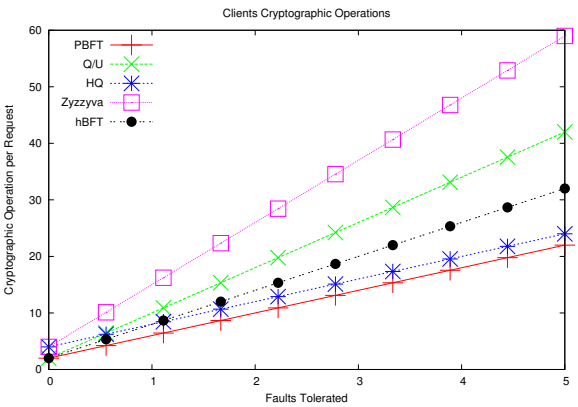
Fig. 9 compares the number of cryptographic operations that the primary and clients perform in normal cases as the number of faults increases. In addition to PBFT, Zyzzyva and *h*BFT, we also include Q/U and HQ, which are two (hybrid) Byzantine quorum protocols. For the performance of a primary with or without batching, as illustrated in Fig. 9(a) and Fig. 9(b), it can be observed that batching greatly reduces the number of cryptographic operations as the number of total replicas increases. For instance, although the number of cryptographic operations of PBFT is high without batching and increases quite fast, the cryptographic overhead is almost the smallest



(a) Bottleneck server, $b = 1$



(b) Bottleneck server, $b = 10$



(c) Client

Fig. 9. Fault scalability using analytical model.

without batching and remains stable as the number of faults increases. Comparably, the number of cryptographic operations of Zyzzyva does not decrease too much without batching. Since both HQ and Q/U are quorum-based protocols, they cannot use batching and work better under low concurrency. *h*BFT achieves the smallest numbers with or without batching.

As illustrated in Fig. 10, as the number of replicas increases, the latency of PBFT increases quickly without batching. With batching, PBFT achieves a more

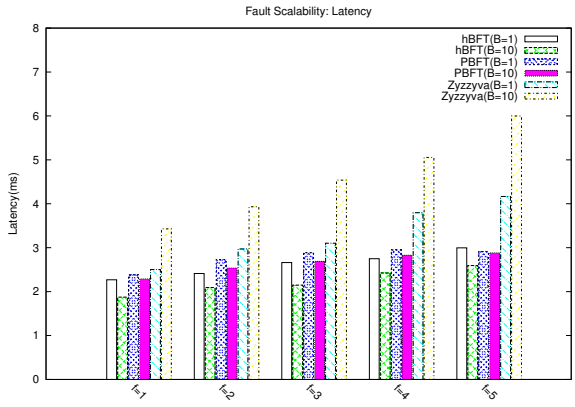


Fig. 10. Fault scalability: latency.

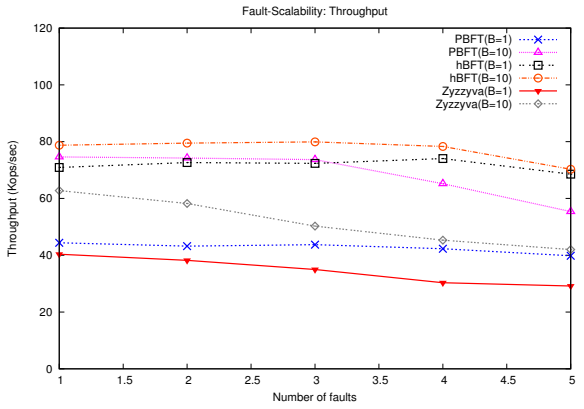


Fig. 11. Fault scalability: throughput.

stable curve. Zyzzyva has higher latency than the other two protocols for each case. On the other hand, the latency of *h*BFT stabilizes and does not grow to a large degree with or without batching. The key factors in the performance are not only the critical paths and the number of cryptographic operations, but also the message complexity. Although Zyzzyva has higher cryptographic overhead, it requires the same number of messages as *h*BFT, explaining why both scale better than PBFT.

Not surprisingly, as shown in Fig. 11, the throughput shows a similar trend with latency. As the system scales, when f is greater than 2, throughput of Zyzzyva obviously decreases, especially without batching. Zyzzyva scales better than PBFT but the performance degrades obviously when f is greater than 4. *h*BFT scales better than both Zyzzyva and PBFT with or without batching. The difference between the numbers of cryptographic operations is still the key to the overall performance. When the number of faults is 5 and assuming b equals 10, PBFT requires 42 MACs without batching and only 6 with batching, Zyzzyva requires 44 MACs without batching and 30.5 with batching, and *h*BFT requires 17 MACs without batching and 3.5 with batching. For systems with high concurrency, PBFT and *h*BFT are preferred and scale well as the number of faults increases.

6.4 A BFT Network File System

This section describes our evaluation of a BFT-NFS service implemented using PBFT [7], Zyzzyva [21], and *h*BFT, respectively. Similarly, in the NFS service, we evaluate the performance of normal cases where a backup server fails.

The NFS service exports a file system, which can then be mounted on a client machine. The replication library and the NFS daemon are called to reach agreement in the order that replicas receive client requests. Once processing is done, replies are sent to the clients. The NFS daemon is implemented using a fixed-size memory-mapped file.

We use the Bonnie++ benchmark [12] to compare our three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. The Bonnie++ benchmark includes sequential input (including per-character and block file reading), sequential output (including per-character and block file writing), and the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that appears random to the file system; (5) stat() random files; (6) delete the files in random order.

We evaluate the performance when a failure occurs at time zero, as detailed in Fig. 12. In addition, up to 20 clients run Bonnie++ benchmark concurrently. The results show that *h*BFT completes every type of operations with lower latency than all of other protocols. The main difference lies on the write operations. This is due to the fact that all the three protocols use read-only optimization, where replicas send reply messages to the clients directly without running the agreement protocol. Compared with NFS-std, *h*BFT only causes 6% overhead while PBFT and Zyzzyva cause 10% and 18% overhead, respectively.

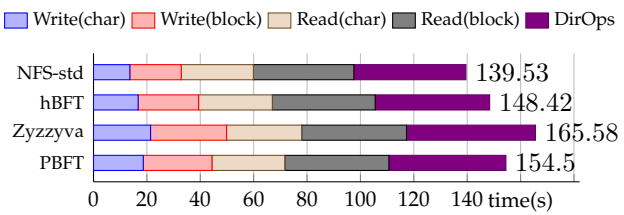


Fig. 12. NFS evaluation with the Bonnie++ benchmark.

7 Conclusion

In this paper, we presented *h*BFT, a hybrid, Byzantine fault-tolerant, replicated state machine protocol with optimal resilience. By re-exploiting speculation, as well a requiring the participation of clients, the

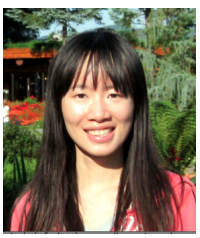
theoretical lower bound for throughput and latency have been achieved for both *fault-free* and *normal cases* in *h*BFT. *h*BFT is a fast protocol that moves some jobs to the clients but can still tolerate faulty clients. We have also proven the safety and liveness properties of *h*BFT and demonstrated how *h*BFT improves on the performance of existing protocols without several of the trade-offs.

ACKNOWLEDGEMENTS

We would like to thank Matt Bishop, Jeff Rowe, Haibin Zhang, Hein Meling, Tiancheng Chang, and Leander Jehi for their helpful comments and contributions to the paper. This research is based on work supported by the National Science Foundation under Grant Number CCF-1018871. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, J. Wylie. Fault-scalable Byzantine fault-tolerant services. *SOSP 2005*, 59–74.
[2] Y. Amir, B. Coan, J. Kirsch, J. Lane. Byzantine replication under attack. *DSN, 2008*, 97–206.
[3] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. *DSN, 2006*,105–114.
[4] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. *In Advances in Cryptology - Eurocrypt 96, Lecture Notes in Computer Science Vol. 1070,* Springer-Verlag, 1996.
[5] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. *In Advances in Cryptology - Crypto 2006, LNCS Vol. 4117,* Springer, 2006.
[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *In Advances in Cryptology - Crypto 96, LNCS Vol. 1109,* Springer, 1996.
[7] M. Castro, and B. Liskov. Practical Byzantine fault tolerance. *OSDI, 1999*,173–186.
[8] T. Chandra, V. Hadzilacos and S. Toueg. The weakest failure detector for solving consensus. *J. ACM 43(4):* 685–722, 1996.
[9] T. Chandra, and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing,* 325–340, 1991.
[10] B. Chun, P. Maniatis, S. Shenker, J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. *SOSP 2007*, 189–204.
[11] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. *NSDI 2009*,153–168.
[12] R. Coker. www.coker.com.au/bonnie++, 2001.
[13] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *OSDI 2006*,177–190.
[14] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. *Ada-Europe 2002,* 24–50.
[15] C. Dwork, and N. Lynch. Consensus in the presence of partial synchrony. *J. ACM 35(2),* 288–323,1988.
[16] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM, 32(2)*,374–382,1985.
[17] R, Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *EuroSys 2010,* 363–376.
[18] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. *HotDep, 2006.*
[19] J. Hendricks, S. Sinnamohideen, G. Ganger, M Reiter. Zzyzx: scalable fault tolerance through Byzantine locking. *DSN 2010*,363–372.
[20] M. Hurfin, M. Raynal. A simple and fast asynchronous consensus protocol. *Distributed Computing 12(4),* 209–223, 1999.
[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SOSP 2007*,45–58.
[22] L. Lamport. The part-time parliament. *Technical Report 49, DEC Systems Research Center*, 1989.
[23] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2): 104–125, 2006.
[24] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2): 79–103, 2006.
[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst., 4(3), 1982.*
[26] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4): 203-213, 1998.
[27] J. Martin, and L. Alvisi. Fast Byzantine consensus. *IEEE Trans. Dependable Sec. Comput.* 3(3): 202-215, 2006.
[28] J. Knight and N. Leveson. An Experimental Evaluation of The Assumption of Independence in MultiVersion Programming. *IEEE Trans. Software Eng.* 12(1): 96–109, 1986.
[29] G. Veronese, M. Correia, A. Bessani, and L. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. *SRDS 2009*,135–144.
[30] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.* 21(3): 236–269, 2003.
[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar An integrated experimental environment for distributed systems and networks. *OSDI 2002*, 255–270.
[32] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. *SOSP, 2003,* 253–267.
[33] P. Zielinski. Low-latency atomic broadcast in the presence of contention. *DISC, 2006,* 505–519.
[34] P. Zielinski. Optimistically terminating consensus: all asynchronous consensus protocols in one framework. *ISPDC 2006,* 24–33.



Sisi Duan is a Ph.D. candidate in Computer Science in the Computer Security Research Lab at the University of California, Davis. Her research interests include fault tolerance, diagnosis, and recovery in distributed systems, publish/subscribe systems, and intrusion detection systems.

Sean Peisert received his Ph.D. in computer science from the University of California, San Diego in 2007. He is currently jointly appointed at Lawrence Berkeley National Laboratory and University of California Davis, and was previously an I3P Research Fellow. His research spans a broad cross section of computer security. He is a senior member of the IEEE.

Karl Levitt received his Ph.D. in electrical engineering from New York University in 1966. He has been a professor of computer science at UC Davis since 1986 and has previously been a program director for the National Science Foundation and director of the Computer Science Laboratory at SRI International. He conducts research in the areas of computer security, automated verification, and software engineering.