Zyzzyva: Speculative Byzantine Fault Tolerance

RAMAKRISHNA KOTLA
Microsoft Research, Silicon Valley
and
LORENZO ALVISI, MIKE DAHLIN, ALLEN CLEMENT, and EDMUND WONG
The University of Texas at Austin

A longstanding vision in distributed systems is to build reliable systems from unreliable components. An enticing formulation of this vision is Byzantine Fault-Tolerant (BFT) state machine replication, in which a group of servers collectively act as a correct server even if some of the servers misbehave or malfunction in arbitrary ("Byzantine") ways. Despite this promise, practitioners hesitate to deploy BFT systems, at least partly because of the perception that BFT must impose high overheads.

In this article, we present Zyzzyva, a protocol that uses speculation to reduce the cost of BFT replication. In Zyzzyva, replicas reply to a client's request without first running an expensive three-phase commit protocol to agree on the order to process requests. Instead, they optimistically adopt the order proposed by a primary server, process the request, and reply immediately to the client. If the primary is faulty, replicas can become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order. This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima and to achieve throughputs of tens of thousands of requests per second, making BFT replication practical for a broad range of demanding services.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Byzantine fault tolerance, speculative execution, replication, output commit

This work is supported in part by NSF grants CNS-0720649, CNS-0509338, and CNS-0411026.
Authors' addresses: R. Kotla, Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043; email: kotla@microsoft.com; L. Alvisi, M. Dahlin, A. Clement, E. Wong, Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233; email: {lorenzo, dahlin, aclement, elwong}@cs.utexas.edu.

DOI 10.1145/1658357.1658358 http://doi.acm.org/10.1145/1658357.1658358

ACM Reference Format:
Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. 2009. Zyzzyva: Speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. 27, 4, Article 7 (December 2009), 39 pages.
DOI = 10.1145/1658357.1658358 http://doi.acm.org/10.1145/1658357.1658358

1. INTRODUCTION

Mounting evidence suggests that real systems must contend not only with simple crashes but also with more complex failures ranging from hardware data corruption [Prabhakaran et al. 2005] to nondeterministic software errors [Yang et al. 2006] and security breaches [Keeney et al. 2005]. Such failures can cause even highly engineered services to become unavailable or to lose data. For example, a single corrupted bit in a handful of messages recently brought down the Amazon S3 storage service for several hours [Amazon 2008], and several well-known email service providers have occasionally lost customer data [Gmail 2006; Hotmail 2004].

Byzantine Fault-Tolerant (BFT) state machine replication is a promising approach to masking many such failures and constructing highly reliable and available services. In BFT replication, $n \geq 3f + 1$ servers collectively act as a *correct* server even if up to f servers misbehave or malfunction in arbitrary ("Byzantine") ways [Lamport et al. 1982; Lamport 1984].

Unfortunately, practitioners hesitate to deploy BFT systems at least partly because of the perception that BFT must impose high overheads. This concern motivates our work, which seeks to answer a simple question: *Can we build a system that tolerates a broad range of faults while meeting the demands of high performance services?*

The basic idea of BFT state machine replication is simple: Clients send requests to a replicated service and the service's distributed agreement protocol ensures that correct servers execute the same requests in the same order [Schneider 1990]. If the service is deterministic, each correct replica thus traverses the same series of states and produces the same reply to each request. The servers send their replies back to the client, and the client accepts a reply that matches across a sufficient number of servers.

Zyzzyva builds on this basic approach, but reduces its cost through *speculation*. As is common in existing BFT state machine replication protocols [Castro and Listov 2002], an elected *primary* server proposes an order on client requests to the other server *replicas*. However, unlike in traditional BFT state machine replication protocols, Zyzzyva replicas then immediately execute requests speculatively, without running an expensive agreement protocol to establish the requests' final order. As a result, if the primary is faulty, correct replicas' states may diverge, and they may send different responses to a client. Nonetheless, Zyzzyva preserves correctness because a correct client detects such divergence and avoids acting on a reply until the reply and sequence of preceding requests are *stable* and guaranteed to be eventually adopted by all correct servers. Thus, applications at correct clients observe the traditional abstraction of a replicated state machine that executes requests in a linearizable [Herlihy and Wing 1990] order.

The challenge in Zyzzyva is ensuring that responses to correct clients become stable. While Zyzzyva ultimately leaves this responsibility to the replicas, a correct client with an outstanding request speeds the process by supplying information that will either cause the request to become stable rapidly or lead to the election of a new primary server, which will be charged with the task of either making pending request stable or face its own demotion.

Essentially, Zyzzyva "rethinks the sync" [Nightingale et al. 2006] for BFT. Instead of enforcing the condition that *a correct server only emits replies that are stable*, Zyzzyva recognizes that this condition is stronger than required. Instead, Zyzzyva enforces the weaker condition: *A correct client only acts on replies that are stable*. This change allows us to move the output commit from the servers to the client, which lets servers to avoid the expensive all-to-all communication required to ensure the stronger condition.

By leveraging this insight, Zyzzyva's replication cost, processing overhead, and end-to-end communication latencies approach their theoretical lower bounds. In practice, Zyzzyva achieves a peak measured throughput of over 86K requests/second on 3.0 GHz Pentium-IV machines during failure-free executions, with only slight throughput reduction, to 82K requests/second, when up to f nonprimary replicas crash, suggesting that Zyzyzva can provide the peace of mind offered by BFT replication for a broad range of demanding services.

1.1 Why Another BFT Protocol?

The past three decades have witnessed remarkable progress in the science of BFT state machine replication. Lamport defined state machine replication in 1978 [Lamport 1978]. Then, in 1982 and 1984, Lamport et al. defined the Byzantine fault model [Lamport et al. 1982] and a BFT state machine replication algorithm for synchronous systems [Lamport 1984]. The approach was significantly refined by Schneider [1990] and Reiter [1995], but the cost of BFT state machine replication protocols and their reliance on synchrony assumptions for safety limited their practicality.

In the last decade, Castro and Listov's [2002] seminal Practical Byzantine Fault Tolerance (PBFT) protocol devised techniques to eliminate expensive signatures and potentially fragile timing assumptions and demonstrated high throughputs of over 10K requests per second. This surprising result jump-started an arms race in which researchers reduced replication costs [Yin et al. 2003], and improved performance [Abd-El-Malek et al. 2005; Cowling et al. 2006; Kotla and Dahlin 2004] of BFT service replication.

Unfortunately, a side-effect of these efforts is that the current state-of-the-art for BFT state machine replication is distressingly complex. In a November 2006 paper describing Hybrid-Quorum replication (HQ replication), Cowling et al. [2006] draw the following conclusions comparing three state-of-the-art protocols (Practical Byzantine Fault Tolerance (PBFT) [Castro and Listov 2002], Query/Update (Q/U) [Abd-El-Malek et al. 2005], and HQ replication [Cowling et al. 2006]).

—"In the regions 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

Table I. Properties of State-of-the-Art and Optimal Byzantine Fault-Tolerant Replication Systems

		PBFT	Q/U	HQ	Zyzzyva	State Machine Repl. Lower Bound
Cost	Total replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	3f+1 [Pease et al. 1980]
	App. replicas	**2f+1**	5f+1	3f+1	**2f+1**	2f+1 [Schneider 1990]
Throughput	MAC ops/request	2+(8f+1)/b	2+8f	4+4f	**2+3f/b**	2[†]
Latency	NW 1-way latencies	4	**2**[*]	4	**3**	2[*] or 3[‡]

These systems tolerate f faults using MACs for authentication [Castro and Listov 2002] and use a batch size of b [Castro and Listov 2002]. Bold entries denote protocols that match known lower bounds or those with the lowest known cost. [†]It is not clear that this trivial lower bound is achievable. [‡]The distributed systems literature typically considers 3 one-way latencies to be the lower bound for agreement on client requests [Dutta et al. 2005; Lamport 2003; Martin and Alvisi 2006]; [*]A delay of 2 one-way latencies is achievable if no concurrency is assumed.

best choice, else HQ is the best since it outperforms [P]BFT with a batch size of 1."

—"Otherwise, [P]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: HQ's throughput decreases more slowly than Q/U's (because of the latter's larger message and processing costs) and [P]BFT's (where eventually batching cannot compensate for the quadratic number of messages)."

Such complexity represents a barrier to the adoption of BFT techniques because it requires a system designer to choose the right technique for a workload and then for the workload not to deviate from expectations.

As Table I indicates, Zyzzyva simplifies the design space of BFT replicated services by approaching the lower bounds in almost every key metric.

With respect to replication cost, Zyzzyva and PBFT match the lower bound, both in the total number of replicas that participate in the protocol and in the number of replicas that must hold copies of application state and execute application requests. Both protocols hold cost advantages of 1.5–2.5 over Q/U and 1.0–1.5 over HQ depending on the number of faults to be tolerated and on the relative cost of replicating application nodes versus agreement nodes.

With respect to throughput, both Zyzzyva and PBFT use batching when load is high and thereby approach the lower bound on the number of authentication operations performed at the bottleneck node, and Zyzzyva approaches this bound more rapidly than PBFT. Unlike state machine replication-based protocols, quorum-based protocols such as Q/U and HQ cannot batch concurrent client requests as they do not have a primary replica funneling all requests to other replicas. As shown in the second row of the Table I, Q/U and HQ's inability to support batching increases the cryptographic overhead per request at the bottleneck node, by factors approaching 5 and 4, respectively, when one fault ($f = 1$) is tolerated and by higher factors in systems that tolerate more faults.

With respect to latency, Zyzzyva executes requests in three one-way message delays, which matches the lower bound for agreeing on a client request [Dutta et al. 2005; Lamport 2003; Martin and Alvisi 2006] and improves upon both

PBFT and HQ. Q/U sidesteps this lower bound by providing a service that is slightly weaker than traditional state machine replication (i.e., by not totally ordering all requests) and by optimizing for cases without concurrent access to any state. This difference presents a chink in Zyzzyva's armor, which Zyzzyva minimizes by matching the lower bound on message delays for full consensus. We believe that Zyzzyva's other advantages over Q/U (fewer replicas, improved throughput via batching, simpler state machine replication semantics, ability to support high-contention workloads) justify this modest additional latency.

With respect to fault scalability [Abd-El-Malek et al. 2005], the metrics that depend on f grow as slowly or more slowly in Zyzzyva as in any other protocol.

Note that as is customary [Abd-El-Malek et al. 2005; Castro and Listov 2002; Cowling et al. 2006; Rodrigues et al. 2001; Yin et al. 2003], Table I compares the protocols' performance during the expected common case of fault-free, timeout-free execution. All protocols are guaranteed to operate correctly in the presence of up to f faults and arbitrary delays, but they can pay significantly higher overheads and latencies in such scenarios [Clement et al. 2009b]. In Section 5.4, we consider the susceptibility of these protocols to faults and argue that Zyzzyva remains the most attractive choice.

2. SYSTEM MODEL

To tolerate a broad range of failures, we assume the Byzantine failure model where faulty nodes (server replicas or clients) may deviate from their intended behavior in arbitrary ways because of problems such as hardware faults, software faults, node misconfigurations, or even malicious attacks. We further assume a strong adversary that can coordinate faulty nodes to compromise the replicated service. However, we assume the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures; we denote a message m signed by principal q's public key as $\langle m \rangle_{\sigma_q}$. Zyzzyva ensures its safety and liveness properties if at most f replicas are faulty, and it assumes a finite client population, any number of which may be faulty.

It makes little sense to build a system that can tolerate Byzantine server replicas and clients but that can be corrupted by an unexpectedly slow node or network link, hence we design Zyzzyva so that its safety properties hold in any asynchronous distributed system where nodes operate at arbitrary speeds and are connected by a network that may corrupt, delay, and fail to deliver messages, or deliver them out of order.

Unfortunately, ensuring both safety and liveness for consensus in an asynchronous distributed system is impossible if any server can crash [Fischer et al. 1985], let alone if servers can be Byzantine. Zyzzyva's liveness, therefore, is ensured only during intervals in which messages sent to correct nodes are processed within some arbitrarily large (but potentially unknown) worst-case delay from when they are sent. This assumption appears easy to meet in practice if broken links are eventually repaired.

Zyzzyva implements a BFT service using state machine replication [Lamport 1984; Schneider 1990]. Traditional state machine replication techniques can be applied only to deterministic services. Zyzzyva copes with the nondeterminism present in many real-world applications such as file systems and databases using standard techniques to abstract the observable application state at the replicas and to resolve nondeterministic choices via the agreement stage [Rodrigues et al. 2001].

If a client of a service issues an erroneous or malicious request, Zyzzyva's job is to ensure the request is processed consistently at all correct replicas; the replicated service itself is responsible for protecting its application state from such erroneous requests. Services typically limit the damage by authenticating clients and enforcing access control, so that, for example, in a replicated file system, if a client tries to write a file without appropriate credentials, all correct replicas process the request by returning an error code.

Services can also limit the damage done by Byzantine clients by maintaining multiple versions of shared data (e.g., snapshots in a file system [Santry et al. 1999; Kotla et al. 2007b]) so that data destroyed or corrupted by a faulty client can be recovered from older versions.

3. PROTOCOL

Zyzzyva is a state machine replication protocol executed by $3f + 1$ replicas and based on three subprotocols: (1) agreement, (2) view change, and (3) checkpoint. The *agreement* subprotocol orders requests for execution by the replicas. Agreement operates within a sequence of *views*, and in each view a single replica, designated as the *primary*, is responsible for leading the agreement subprotocol. The *view change* subprotocol coordinates the election of a new primary when the current primary is faulty or the system is running slowly. The *checkpoint* subprotocol limits the state that must be stored by replicas and reduces the cost of performing view changes.

Figure 1 shows the communication pattern for a single instance of Zyzzyva's agreement subprotocol. In the fast, no-fault case (Figure 1(a)), a client simply sends a request to the primary, the primary forwards the request to the replicas, and the replicas execute the request and send their responses to the client.

A request *completes* at a client when the client has a sufficient number of matching responses to ensure that all correct replicas will eventually execute the request and all preceding requests in the same order, thus guaranteeing that all correct replicas process the request in the same way, issue the same reply, and transition to the same subsequent system state. To allow a client to determine when a request completes, a client receives from replicas *responses* that include both an application-level *reply* and the *history* on which the reply depends. The *history* is the sequence of all requests executed by a replica prior to and including this request.

As Figure 1 illustrates, a request completes at a client in one of two ways. First, if the client receives $3f + 1$ matching responses (Figure 1(a)), then the client considers the request complete and acts on it. Second, if the client receives between $2f + 1$ and $3f$ matching responses (Figure 1(b)), then the client gathers $2f + 1$ matching responses in a *commit certificate* that it distributes to the replicas. A commit certificate includes cryptographic proof that $2f + 1$ servers agree on a linearizable order for the request and all preceding requests, and

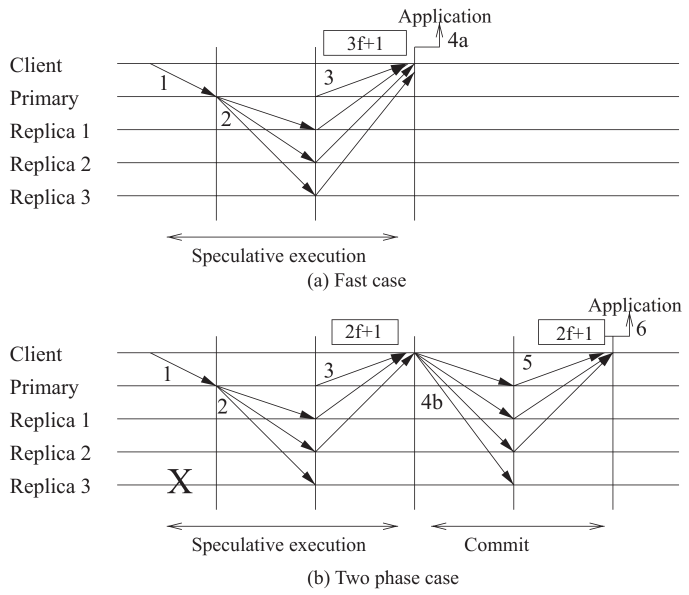


Fig. 1. Protocol communication pattern for agreement within a view for: (a) the fast case; and (b) the two-phase faulty-replica case. The numbers refer to the main steps of the protocol in the text.

successfully storing a commit certificate to $2f + 1$ servers (and thus at least $f + 1$ correct servers) ensures that no other ordering can muster a quorum of $2f + 1$ servers to contradict this order. Therefore, once Once $2f + 1$ replicas acknowledge receiving a commit certificate, the client considers the request complete and acts on the corresponding reply.

Zyzzyva then ensures the following safety condition.

SAF If a request with sequence number n and history h_n completes, then any request that completes with a higher sequence number $n' \geq n$ has a history $h_{n'}$ that includes h_n as a prefix.

If fewer than $2f + 1$ responses match, then to ensure liveness the client retransmits the request to all replicas, which then begin waiting for the primary to order the retransmitted request. If a correct replica sees that the primary is ordering the request too slowly or inconsistently, it starts suspecting that the primary is faulty. If a sufficient number of replicas suspect that the primary is faulty, then a view change occurs and a new primary is elected.

Assuming eventual synchrony[1] [Dwork et al. 1988], Zyzzyva then ensures the following liveness condition.

LIV Any request issued by a correct client eventually completes.

For the sake of clarity, in the rest of this section we describe and outline the proof of correctness for an unoptimized version of Zyzzyva that relies on

[1]In practice eventual synchrony can be achieved by using exponentially increasing timeouts [Castro and Listov 2002].

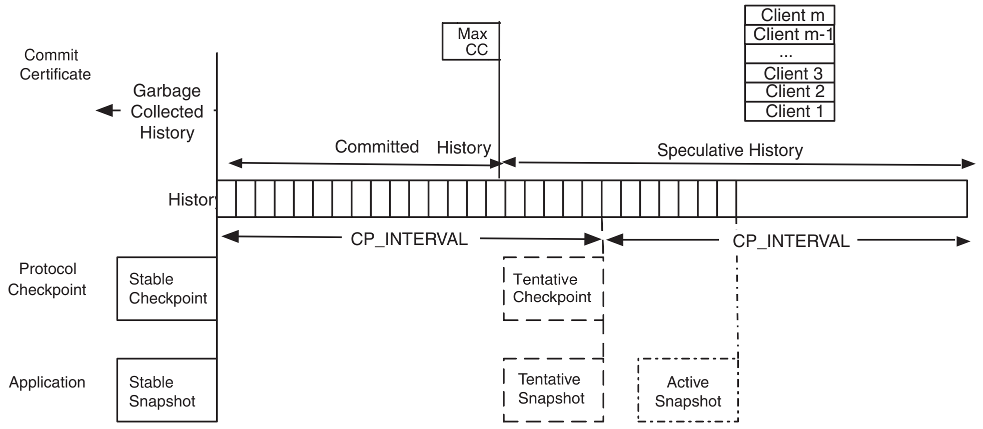


Fig. 2. State maintained at each replica.

digital signatures. In Section 4 we describe a number of optimizations, all implemented in our prototype, that reduce encryption costs by replacing digital signatures with Message Authentication Codes (MACs), improve throughput by batching requests, reduce the impact of lost messages by caching out-of-order messages, improve read performance by optimizing read-only requests, reduce bandwidth by having most replicas send hashes rather than full replies, reduce overheads by including MACs only for a preferred quorum, and improve performance in the presence of faulty nodes using a commit phase optimization. In Section 4.1 we discuss Zyzzyva5, a variation of the protocol that requires $5f + 1$ agreement replicas but that completes in three one-way message exchanges as in Figure 1(a) even when up to f nonprimary replicas are faulty.

3.1 Node State and Checkpoint Subprotocol

To ground our discussion, we begin by discussing the state maintained by each replica as summarized by Figure 2. Each replica i maintains an ordered *history* of the requests it has executed and a copy of the *max commit certificate*, the commit certificate seen by i that covers the largest prefix of i's stored history. The history up to and including the request with the highest sequence number covered by this commit certificate is the *committed history*, and the history that follows is the *speculative history*. We say that a commit certificate has sequence number n if n is the highest sequence number of any request in the committed history.

A replica constructs a checkpoint every *CP_INTERVAL* requests. A replica maintains one *stable checkpoint* and a corresponding *stable application state snapshot*, and it may store up to one *tentative checkpoint* and corresponding *tentative application state snapshot.* The process by which a tentative checkpoint and application state become stable is similar to the one used by earlier BFT protocols [Castro and Listov 2002; Cowling et al. 2006; Kotla and Dahlin 2004; Rodrigues et al. 2001; Yin et al. 2003] with the exception that Zyzzyva adds

Table II. Labels Given to Fields in Messages

Label	Meaning
c	Client ID
CC	Commit Certificate
d	Digest (cryptographic 1-way hash) of client request message: $d = H(m)$
i, j	Server IDs
h_n	History through sequence number n encoded as cryptographic 1-way hash: $h_n = H(h_{n-1}, d)$
m	Message containing client request
max_n	Max sequence number accepted by replica
n	Sequence number
ND	Selection of nondeterministic values needed to execute a request
o	Operation requested by client
OR	Order Request message
POM	Proof Of Misbehavior
r	Application reply to a client operation
t	Timestamp assigned to an operation by a client
v	View number

an additional all-to-all communication among replicas to commit the tentative history of requests included in the checkpoint, as explained in Appendix A.

To bound the size of history, a correct replica: (1) truncates the history before the committed checkpoint and (2) blocks processing of new requests after processing 2 × *CP_INTERVAL* requests since the last committed checkpoint.

Finally, in order to support execute exactly-once semantics, each replica maintains a *response cache* containing a copy of the latest ordered request from, and corresponding response to, each client.

3.2 Agreement Subprotocol: Fast Case

We detail Zyzzyva's agreement subprotocol by considering three cases: (1) the *fast case* when all nodes act correctly and no timeouts occur, (2) the *two-phase case* that can occur when a nonprimary replica is faulty or some timeouts occur, and (3) the *view change* case that can occur when the primary is faulty or more serious timeouts occur. Table II summarizes the labels we give fields in messages. Most readers will be happier if on their first reading they skip the text marked *additional details*.

Figure 1(a) illustrates the basic flow of messages in the fast case. We trace these messages through the system to explain the protocol, with the numbers in the figure corresponding to the numbers of major steps in the text. As the figure illustrates, the fast case proceeds in four major steps.

1. Client sends request to the primary.

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

4a. Client receives $3f+1$ matching responses and completes the request.

3.2.1 *Message Processing Details.* To ensure correctness, the messages are carefully constructed to carry sufficient information to link these actions with one another and with past system actions. We now detail the contents of each message and describe the steps each node takes to process each message.

1. Client sends request to the primary.

A client c requests an operation o be performed by the replicated service by sending a message $\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}$ to the replica it believes to be the primary (i.e., the primary for the last response received by the client).

Additional details. If the client guesses the wrong primary, the retransmission mechanisms discussed in step 4c forward the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests [Castro and Listov 2002].

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

A view's primary has the authority to propose the order in which the system should execute requests. It does so by producing ORDER-REQ messages in response to client REQUEST messages.

In particular, when When the primary p receives message $m = \langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}$ from client c, the primary assigns to the request a sequence number n in the current view v and relays a message $\langle\langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle_{\sigma_p}, m\rangle$ to the nonprimary (backup) replicas, where n and v indicate the proposed sequence number and view number for m, digest $d = H(m)$ is the cryptographic one-way hash of m, $h_n = H(h_{n-1}, d)$ is a cryptographic hash summarizing the history, and ND is a set of values for nondeterministic application variables (time in file systems) required for executing the request.

Additional details. The primary only takes the preceding actions if $t > t_c$, where t_c is the highest timestamp previously received from c.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

When a replica receives an ORDER-REQ message, it optimistically assumes that the primary is correct and that other correct replicas will receive the same request with the same proposed order. It therefore speculatively executes requests in the order proposed by the primary and produces a SPEC-RESPONSE message that it sends to the client.

In particular, upon Upon receipt of a message $\langle\langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle_{\sigma_p}, m\rangle$ from the primary p, replica i accepts the ordered request if m is a well-formed

REQUEST message, d is a correct cryptographic hash of m, v is the current view, $n = max_n + 1$ where max_n is the largest sequence number in i's history, and $h_n = H(h_{n-1}, d)$. Upon accepting the message, i appends the ordered request to its history, executes the request using the current application state to produce a reply r, and sends to c a message $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$, where $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle_{\sigma_p}$.

Additional details. A replica may only accept and speculatively execute requests in sequence number order, but message loss or a faulty primary can introduce holes in the sequence number space. Replica i discards the ORDER-REQ message if $n \leq max_n$. If $n > max_n + 1$, then i discards the message, sends a message $\langle \text{FILL-HOLE}, v, max_n + 1, n, i\rangle_{\sigma_i}$ to the primary, and starts a timer. Upon receipt of a message $\langle \text{FILL-HOLE}, v, k, n, i\rangle_{\sigma_i}$ from replica i, the primary p sends a $\langle\langle \text{ORDER-REQ}, v, n', h_{n'}, d, ND\rangle_{\sigma_p}, m'\rangle$ to i for each request m' that p ordered in the sequence number interval $k \leq n' \leq n$ during the current view; the primary ignores fill-hole requests from other views. If i receives the valid ORDER-REQ messages needed to fill the holes, it cancels the timer. Otherwise, the replica i broadcasts the FILL-HOLE message to all other replicas and initiates a view change when the timer fires. Any replica j that receives a FILL-HOLE message from i sends the corresponding ORDER-REQ message, if it has received one. If, in the process of filling-in holes in the replica sequence, replica i receives conflicting ORDER-REQ messages, then the conflicting messages form a proof of misbehavior as described in protocol step 4d.

4a. Client receives $3f+1$ matching responses and completes the request.

Upon receiving $3f+1$ distinct messages $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$, where i identifies the replica issuing the response, a client determines if they match. SPEC-RESPONSE messages from distinct replicas *match* if they have identical v, n, h_n, $H(r)$, c, t, OR, and r fields.

In the absence of faults and timeouts, all $3f + 1$ responses will match, and $3f + 1$ matching responses suffice to guarantee that it is safe for the client to rely on the corresponding reply. In particular, $3f + 1$ matching responses guarantee that, even in the event of a view change, the position of the request in the history of correct replicas will not change. The reason, once again, comes down to intersections between quorums of replicas. In particular, a client can receive $3f+1$ matching responses only if all correct replicas (which are at least $2f+1$) send matching responses. As we will see in Section 3.5, the view change subprotocol invoked in the event of a primary failure determines the correct state of the service by collecting the histories of $2f + 1$ responsive servers, but any group of $2f + 1$ servers must include at most f faulty servers and at least $f+1$ correct servers and thus, correct servers are always able to out-vote faulty servers to keep a history consistent with the response seen by the client.

3.3 Agreement Subprotocol: Two-Phase Case

If the network, primary, or some replicas are slow or faulty, the client c may not receive matching responses from all $3f + 1$ replicas. The two-phase case applies when the client receives between $2f + 1$ and $3f$ matching responses. As Figure 1(b) illustrates, steps 1–3 occur as described before, but step 4 is different.

4b. Client receives between $2f+1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

The commit certificate is a cryptographic proof that a majority of correct servers agree on the ordering of requests up to and including the client's request. Protocol steps 5 and 6 complete the second phase of agreement by ensuring that enough servers have this proof.

5. Replica receives a COMMIT message from a client containing a commit certificate and acknowledges with a LOCAL-COMMIT message.

6. Client receives LOCAL-COMMIT messages from $2f + 1$ replicas and completes the request.

3.3.1 *Message Processing Details.* Again, the details of message construction and processing are designed to allow clients and replicas to link the system's actions together into a single linearizable history.

4b. Client receives between $2f+1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

A client c sets a timer when it first issues a request. When this timer expires, if c has received matching speculative responses from between $2f + 1$ and $3f$ replicas, then c has a proof that a majority of correct replicas agree on the order in which the request should be processed. Unfortunately, the replicas themselves are unaware of this quorum of matching responses; they only know of their local decision, which may not be enough to guarantee that the request completes in this order.

Figure 3 illustrates the problem. A client receives $2f+1$ matching speculative responses indicating that a request *req* was executed as the nth operation in view v. Let these responses come from $f + 1$ correct servers $\mathcal{C}$ and f faulty servers $\mathcal{F}$ and assume the remaining f correct servers $\mathcal{C}'$ received an ORDER-REQ message from a faulty primary proposing to execute a different request req' at sequence number n in view v. Suppose a view change occurs at this time. The view change subprotocol must determine what requests were executed with what sequence numbers in view v so that the state in view $v + 1$ is consistent with the state in view v. Furthermore, since up to f replicas may be faulty, the view change subprotocol must be able to complete using information from only $2f + 1$ replicas. Suppose now that the $2f + 1$ replicas contributing state to a view change operation are one correct server from $\mathcal{C}$, f faulty servers from $\mathcal{F}$, and f correct but misled servers from $\mathcal{C}'$. In this case, only one of the replicas initializing the new view is guaranteed to vote to execute *req* as operation n in the new view, while as many as $2f$ replicas may vote to execute req' in that

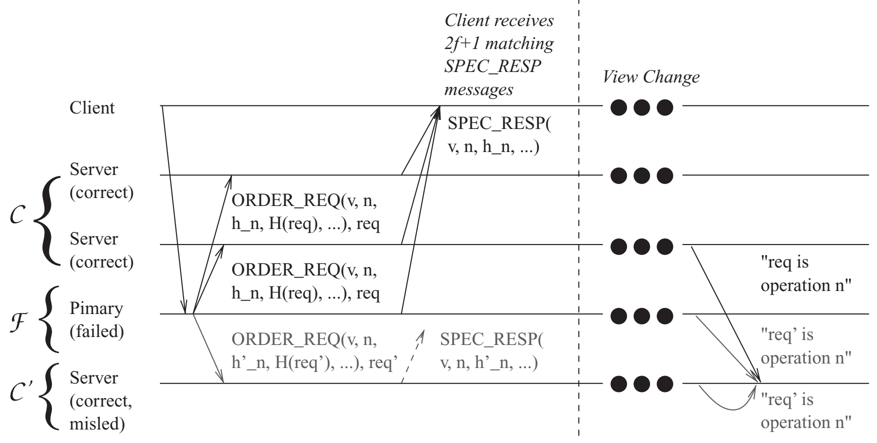

Client receives 2f+1 matching SPEC_RESP messages
View Change
Client
Server (correct)
Server (correct)
Pimary (failed)
Server (correct, misled)
$\mathcal{C}$
$\mathcal{F}$
$\mathcal{C}'$
SPEC_RESP(v, n, h_n, ...)
ORDER_REQ(v, n, h_n, H(req), ...), req
ORDER_REQ(v, n, h_n, H(req), ...), req
ORDER_REQ(v, n, h'_n, H(req'), ...), req'
SPEC_RESP(v, n, h'_n, ...)
"req is operation n"
"req' is operation n"
"req' is operation n"

Fig. 3. Example of a problem that could occur if a client were to rely on just $2f + 1$ matching responses without depositing a commit certificate with the servers.

position. Thus, the system cannot ensure that view $v + 1$'s state reflects the execution of *req* as the operation with sequence number n.

Before client c can rely on this response, it must take additional steps to ensure the response's stability. The client therefore sends a message $\langle$COMMIT, c, $CC\rangle_{\sigma_c}$, where CC is a commit certificate consisting of a list of $2f + 1$ replicas, the replica-signed portions of the $2f + 1$ matching SPEC-RESPONSE messages from those replicas, and the corresponding $2f + 1$ replica signatures.

Additional details. CC contains $2f + 1$ signatures on the SPEC-RESPONSE message and a list of $2f + 1$ nodes, but since all the responses received by c from replicas are identical, c only needs to include one replica-signed portion of the SPEC-RESPONSE message. Also note that, for efficiency, CC does not include the body r of the reply but only the hash $H(r)$.

5. Replica receives a COMMIT message from a client containing a commit certificate and acknowledges with a LOCAL-COMMIT message.

When a replica i receives a message $\langle$COMMIT, c, $CC\rangle_{\sigma_c}$ containing a valid commit certificate CC proving that a request should be executed with a specified sequence number and history in the current view, the replica first ensures that its local history is consistent with the one certified by CC. If so, replica i stores CC if CC's sequence number exceeds the stored *max commit* certificate's sequence number and sends a message $\langle$LOCAL-COMMIT, $v, d, h, i, c\rangle_{\sigma_i}$ to c.

Additional details. If the local history simply has holes encompassed by CC's history, then i fills them as described in protocol step 3. If, however, the two histories contain different requests for the same sequence number, then i initiates the view change subprotocol. Note that as the view change protocol executes, correct replicas converge on a single common history, and those replicas whose local state reflect the "wrong" history (e.g., because they speculatively executed the "wrong" requests) restore their state from a cryptographically signed distributed global stable state.

6. Client receives LOCAL-COMMIT messages from $2f + 1$ replicas and completes the request.

The client resends the COMMIT message until it receives corresponding LOCAL-COMMIT messages from $2f + 1$ distinct replicas. The client then considers the request to be complete and delivers the reply r to the application.

$2f + 1$ LOCAL-COMMIT messages suffice to ensure that a client can rely on a response. In particular, at least $f + 1$ correct servers store a commit certificate for the response, and since any commit or view change requires participation by at least $2f + 1$ of the $3f + 1$ servers, any subsequent committed request or view change includes information from at least one correct server that holds this commit certificate. Since the commit certificate includes $2f + 1$ signatures vouching for the response, even a single correct server can use the commit certificate to convince all correct servers to accept this response, including the application reply and the history.

Additional details. When the client first sends the COMMIT message to the replicas it starts a timer. If this timer expires before the client receives $2f + 1$ LOCAL-COMMIT messages, then the client moves on to protocol steps described in Section 3.4.

3.3.2 *Client Trust.* At first glance, it may appear imprudent to rely on clients to transmit commit certificates to replicas (4b): what if a faulty client sends an altered commit certificate (threatening safety) or fails to send a commit certificate (imperiling liveness)?

Safety is ensured even if clients are faulty because commit certificates are authenticated by $2f + 1$ replicas. If a client alters a commit certificate, correct replicas will ignore it.

Liveness is ensured *for correct clients* because commit certificates are cumulative: Successfully storing a commit certificate for request n at $2f + 1$ replicas commits those replicas to a linearizable total order for all requests up to request n. So, if a faulty client fails to deposit a commit certificate, that client may not learn when its request completes, and a replica whose state has diverged from its peers may not immediately discover this fact. However, if at any future time a correct client issues a request, that request (and a linearizable history of earlier requests on which it depends) will either: (i) complete via $3f + 1$ matching responses (4a), (ii) complete via successfully storing a commit certificate at $2f + 1$ replicas (4b–6), or (iii) trigger a view change (4c or 4d that shortly follow).

3.4 Agreement Subprotocol: View Change Case

Cases 4a and 4b allow a client c's request to complete with $2f + 1$ to $3f + 1$ matching responses. However, if the primary or network is faulty, c may not receive matching SPEC-RESPONSE or LOCAL-COMMIT messages from even $2f + 1$ replicas. Cases 4c and 4d therefore ensure that a client's request either completes in the current view or that a new view with a new primary is initiated. In particular, case 4c is triggered when a client receives fewer than

$2f + 1$ matching responses and case 4d occurs when a client receives responses indicating inconsistent ordering by the primary.

4c. Client receives fewer than $2f + 1$ matching SPEC-RESPONSE messages and resends its request to all replicas, which forward the request to the primary in order to ensure the request is assigned a sequence number and eventually executed.

A client sets a second timer when it first issues a request. If the second timer expires before the request *completes*, the client suspects that the primary may not be ordering requests as intended, so it resends its REQUEST message through the remaining replicas so that they can track the request's progress and, if progress is not satisfactory, initiate a view change. This case can be understood by examining the behavior of a nonprimary replica and of the primary.

Replica. When nonprimary replica i receives a message $\langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}$ from client c, then if the request has a higher timestamp than the currently cached response for that client, i sends a message $\langle \text{CONFIRM-REQ}, v, m, i\rangle_{\sigma_i}$ where $m = \langle \text{REQUEST},o,t,c\rangle_{\sigma_c}$ to the primary p and starts a timer. If the replica accepts an ORDER-REQ message for this request before the timeout, it processes the ORDER-REQ message as described earlier. If the timer fires before the primary orders the request, the replica initiates a view change.

Primary. Upon receiving the message $\langle \text{CONFIRM-REQ},v,m,i\rangle_{\sigma_i}$ from replica i, the primary p checks the client's timestamp for the request. If the request is new, p sends a new ORDER-REQ message using a new sequence number as described in step 2.

Additional details. If replica i does not receive the ORDER-REQ message from the primary, the replica sends the CONFIRM-REQ message to all other replicas. Upon receipt of a CONFIRM-REQ message from another replica j, replica i sends the corresponding ORDER-REQ message it received from the primary to j; if i did not receive the request from the client, i acts as if the request came from the client itself. To ensure eventual progress, a replica doubles its current timeout in each new view and resets it to a default value if a view succeeds in executing a request.

Additionally, to retain exactly-once semantics, replicas maintain a cache that stores the reply to each client's most recent request. If a replica i receives a request from a client and the request matches or has a lower client-supplied timestamp than the currently cached request for client c, then i simply resends the cached response to c. Similarly, if the primary p receives an old client request from replica i, p sends to i the cached ORDER-REQ message for the most recent request from c. Furthermore, if replica i has received a commit certificate or stable checkpoint for a subsequent request, then the replica sends a LOCAL-COMMIT to the client even if the client has not transmitted a commit certificate for the retransmitted request.

4d. Client receives responses indicating inconsistent ordering by the primary and sends a proof of misbehavior to the replicas, which initiate a view change to oust the faulty primary.

If client c receives a pair of SPEC-RESPONSE messages containing valid messages $OR = \langle\text{ORDER-REQ}, v, n, h_n, d, ND\rangle_{\sigma_j}$ for the same request ($d = H(m)$) in the same view v with differing sequence number n or history h_n or ND, then the pair of ORDER-REQ messages constitutes a Proof Of Misbehavior (*POM*) [Aiyer et al. 2005] against the primary. Upon receipt of a *POM*, c sends a message $\langle\text{POM}, v, POM\rangle_{\sigma_c}$ to all replicas. Upon receipt of a valid POM message, a replica initiates a view change and forwards the POM message to all other replicas.

For completeness, note that cases 4b and 4c are not exclusive of 4d; a client may receive messages that are both sufficient to complete a request and also a proof of misbehavior against the primary.

3.5 View Changes

Fast agreement and speculative execution have profound effects on Zyzzyva's view change subprotocol. First, we highlight the differences between the Zyzzyva view change subprotocol and that of previous systems for completeness. We then explain the exact message exchange and processing details in Section 3.5.3.

The view change subprotocol must elect a new primary and guarantee that it will not introduce any changes in a history that has already completed at a correct client. To maintain this safety property, traditional view change subprotocols [Castro and Listov 2002; Cowling et al. 2006; Kotla and Dahlin 2004; Rodrigues et al. 2001; Yin et al. 2003] require a correct replica that commits to a view change to stop accepting messages other than CHECKPOINT, VIEW-CHANGE, and NEW-VIEW messages. Also, to prevent faulty replicas from disrupting the system, a view change subprotocol should never remove a primary unless at least one correct replica commits to the view change. Hence, a correct replica traditionally commits to a view change if either: (a) it observes the primary to be faulty or (b) it has a proof that $f + 1$ replicas have committed to a view change. On committing to a view change a correct replica sends a signed VIEW-CHANGE message that includes the new view, the sequence number of the replica's latest stable checkpoint (together with a proof of its stability), and the set of prepare certificates (the equivalent of commit certificates in Zyzzyva) collected by the replica.

The traditional view change completes when the new primary, using $2f + 1$ VIEW-CHANGE messages from distinct replicas, computes the history of requests that all correct replicas must adopt to enter the new view. The new primary includes this history, with a proof of validity, in a signed NEW-VIEW message that it broadcasts to all replicas.

Zyzzyva maintains the overall structure of the traditional protocol, but it departs in two ways that together allow clients to accept a response before any replicas know that the request has been committed and allow the replicas to commit to a response after two phases instead of the traditional three.

(1) First, to ensure liveness, Zyzzyva strengthens the condition under which a correct replica commits to a view change by adding a new "I hate the primary" phase to the view change subprotocol. We explain the need for and details of this addition shortly by considering *the case of the missing phase.*

(2) Second, to guarantee safety, Zyzzyva weakens the condition under which a request appears in the history included in the NEW-VIEW message. We explain the need for and details of this change later by considering *the case of the uncommitted request.*

3.5.1 *The Case of the Missing Phase.* As Figure 1 shows, Zyzzyva's agreement protocol guarantees that every request that completes within a view does so after at most two phases. This property may appear surprising to the reader familiar with PBFT. If we view a correct client that executes step 4b of Zyzzyva as implementing a broadcast channel between replicas, then Zyzzyva's communication pattern maps to only two of PBFT's three phases, one where communication is primary-to-replicas *(preprepare)* and the second involving all-to-all exchanges (either *prepare* or *commit)*. Where did the third phase go? And why is it there in the first place?

The answer to the second question lies in the subtle dependencies between the agreement and view change subprotocols. No replicated service that uses the traditional view change protocol can be live without an agreement protocol that includes both the *prepare* and *commit* phases.[2] To see how this constraint applies to BFT state machine replication-based protocols, consider a scenario with f faulty replicas, one of them the primary, and suppose the faulty primary causes f correct replicas to commit to a view change and stop sending messages in the view. In this situation, a client request may only receive $f + 1$ responses from the remaining correct replicas, not enough for the request to complete in either the first or second phase; and because fewer than $f + 1$ replicas demand a view change, there is no opportunity to regain liveness by electing a new primary.

The third phase of traditional BFT agreement breaks this stalemate: By exchanging what they know, the remaining $f + 1$ correct replicas either gather the evidence necessary to complete the request after receiving only $f + 1$ matching responses or determine that a view change is necessary.

Back to the first question: How does Zyzzyva avoid the third phase in the agreement subprotocol? The insight is that what compromises liveness in the previous scenario is that the traditional view change protocol lets correct replicas commit to a view change and become silent in a view without any guarantee that their action will lead to the view change. Instead, in Zyzzyva, a correct replica does not abandon the current view unless it is guaranteed that every other correct replica will do the same, forcing a new view and a new primary.

To ensure this property, the Zyzzyva view change subprotocol adds an additional phase to strengthen the conditions under which a replica stops participating in the current view. In particular, a correct replica i that suspects the primary of view v continues to participate in the view, but expresses its vote of no-confidence in the primary by multicasting to all replicas the message $\langle \text{I-HATE-THE-PRIMARY}, v\rangle_{\sigma_i}$. If i receives $f + 1$ votes of no confidence in v's primary, then it commits to a view change: It becomes silent, and multicasts to all

[2]Unless a client can unilaterally initiate a view change. This option is unattractive in our setting where clients can be Byzantine.

replicas a VIEW-CHANGE message that contains a proof that $f + 1$ replicas have no confidence in the primary for view v. A correct replica that receives a valid VIEW-CHANGE message joins in the mutiny and commits to the view change. As a result, Zyzzyva's view change protocol ensures that if a correct replica commits to a view change in view v, eventually all correct replicas will. In effect, Zyzzyva shifts the costs needed to deal with a faulty primary from the critical path (the agreement protocol) to the view change subprotocol, which is expected to be run only when the primary is faulty.

3.5.2 *The Case of the Uncommitted Request.* Zyzzyva replicas may never learn the outcome of the agreement protocol: Only clients may know when a request has completed. How do Zyzzyva replicas identify a safe history prefix for a new view?

There are two ways in which a request r and its history may complete in Zyzzyva. Let us first consider the least problematic from the perspective of a view change: A request r completes because a client receives $2f + 1$ LOCAL-COMMIT messages, implying that at least $f + 1$ correct replicas have stored a commit certificate for r. Traditional view change protocols already handle this case: The standard VIEW-CHANGE message sent by a correct replica includes all commit certificates known to the replica since the latest stable checkpoint. The new primary includes in the NEW-VIEW message all commit certificates that appear in any of the $2f + 1$ valid VIEW-CHANGE messages it receives: at least one of those VIEW-CHANGE messages must contain a commit certificate for r.

The other case is more challenging: If r completes because the client receives $3f + 1$ matching speculative responses, then no correct replica will have a commit certificate for r. We handle this case by modifying the view change subprotocol in two ways. First, correct replicas add to the information included in their VIEW-CHANGE message all ORDER-REQ messages (without the corresponding client request) received since the latest stable checkpoint. Second, a correct new primary extends the history to be adopted in the new view to include all requests with an ORDER-REQ message containing a sequence number higher than the largest sequence number in any commit certificate that appears in at least $f + 1$ of the $2f + 1$ VIEW-CHANGE messages the new primary collects.

This change weakens the conditions under which a request ordered in one view can appear in a new view: We no longer require a commit certificate but also allow a sufficient number of ORDER-REQ messages to support a request's ordering. This change ensures that the protocol continues to honor ordering commitments for any request that completes when a client gathers $3f + 1$ matching speculative responses.

Notice that this change may have the side-effect of assigning an order to a request that has not yet completed in the previous view. In particular, a curiosity of the protocol is that, depending on which set of $2f + 1$ VIEW-CHANGE messages the primary uses, it may, for a given sequence number, find different requests with $f + 1$ ORDER-REQ messages. This curiosity, however, is benign and cannot cause the system to violate safety. In particular, there can be two such candidate requests for the same sequence number only if at least one correct replica supports each of the candidates. In such a case, neither of the candidates could have completed by having a client receive $3f + 1$ matching responses, and the system can safely assign either (or neither) request to that sequence number.

3.5.3 *View Change Subprotocol.* The Zyzzyva view change subprotocol proceeds as follows.

VC1. Replica initiates the view change by sending an accusation against the primary to all replicas.

Replica i initiates a view change by sending $\langle \text{I-HATE-THE-PRIMARY}, v\rangle_{\sigma_i}$ to all replicas, indicating that the replica is dissatisfied with the behavior of the current primary. In previous protocols, this message would indicate that replica i is no longer participating in the current view. In Zyzzyva, this message is only a hint that i would like to change views. Even after issuing the message, i continues to participate faithfully in the current view.

VC2. Replica receives $f + 1$ accusations that the primary is faulty and commits to the view change.

Replica i commits to a view change into view $v+1$ by sending an indictment of the current primary, consisting of $\langle \text{I-HATE-THE-PRIMARY}, v\rangle_{\sigma_j}$ from $f+1$ distinct replicas j, and the message $\langle \text{VIEW-CHANGE}, v+1, s, C, CC, O, i\rangle_{\sigma_i}$ to all replicas. O is i's ordered request history since the last stable checkpoint with sequence number s. C is the proof of the last stable checkpoint consisting of $2f+1$ checkpoint messages. CC is the most recent commit certificate for a request since the last view change.

VC3. Replica receives $2f + 1$ view change messages.

Primary. Upon receipt of $2f + 1$ valid VIEW-CHANGE messages (including its own), the new primary p constructs the message $\langle \text{NEW-VIEW}, v+1, P, G, \sigma_p\rangle_{\sigma_p}$, where P is the set containing valid VIEW-CHANGE messages received by the new primary for view $v+1$. G is the ordered request history computed by the new primary using P.

Backup. A backup replica starts a timer upon receipt of $2f + 1$ valid VIEW-CHANGE messages (including its own). If the backup replica does not receive a valid NEW-VIEW message from the new primary before the timer expires, then the replica initiates a view change into view $v+2$. The length of the timer in the new view grows exponentially with the number view changes that fail in succession.

Additional details. The new primary computes G in new view $v+1$ as follows.

—The primary determines *min-s* as the latest stable checkpoint in the view change messages in P, *max-cc* as the highest sequence number of a committed certificate CC, *max-r* as the highest sequence number of a request that is not committed but (potentially) completed at a client on the fast path, and *max-s* as the highest sequence number in some ordered request history log O, where $min\text{-}s \le max\text{-}cc \le max\text{-}r \le max\text{-}s$.

—*Committed Requests*. The primary inserts $\langle\langle$ORDER-REQ, $v+1$, n, h_n, d, $ND\rangle_{\sigma_p}$, $m\rangle$ into G by copying requests from the ordered history log of the replica that sent the VIEW-CHANGE with *max-cc*, for every sequence number n where $min\text{-}s < n \le max\text{-}cc$. As a performance optimization, we do not include request m in G, but fetch it later if it is not found in the replicas' local order history log.

—*Uncommitted But Potentially Completed Requests*. The primary inserts $\langle\langle$ORDER-REQ, $v+1$, n, h_n, d, $ND\rangle_{\sigma_p}$, $m\rangle$ in G for every sequence number n between *max-cc*+1 and *max-r* such that the following conditions are true: (1) Request n is present in ordered request history logs of at least $f+1$ distinct replicas with matching sequence number n, request history digest h_n, request digest d, and the ND set, and (2) $h_n = H(h_{n-1}, d)$.

In PBFT, requests that are not committed are discarded; Zyzzyva instead retains them as long as they meet the preceding conditions. Zyzzyva does so to avoid the risk of losing, during a view change, requests that completed when a client received $3f+1$ matching responses, but are not committed. Note that at least $f+1$ correct replicas out of the $3f+1$ replicas that sent matching responses are guaranteed to contribute to the state collected by the new primary during the view change protocol, thereby ensuring that all complete requests, whether or not they are committed, are passed on to the next view.

—*Requests That are Guaranteed Not to Have Completed.* If a request's sequence number is at most max-s but the request does not meet the previous two conditions, then the request has definitely not completed. As in PBFT, the primary replaces such requests with a null request by creating $\langle\langle$ORDER-REQ, $v+1$, n, h_n, d^{Null}, $Null\rangle_{\sigma_p}$, $Null\rangle$ for all n such that $max\text{-}r{+}1 \le n \le max\text{-}s$. A null request goes through the protocol as a regular request but is treated as a noop when executed.

VC4. Replica receives a valid new view message, reconciles its local state, and changes to the new view.

Upon receipt of a new view message, replicas (including new primary) reconcile their local state with the state received in the new view message, change to the new view, and start processing messages in the new view.

Additional details.

Primary. The primary reconciles its local state by comparing its local history log O with that of G and using the following steps. Let *max-l* be the latest request in O.

—If *max-l* is less than *min-s*, then the primary inserts the checkpoint certificate with sequence number *min-s* in its history log, discards information from the request history log, and copies ordered requests from G to O starting from *min-s*+1. It also acquires an application-level snapshot for *min-s* by contacting the replicas in the checkpoint certificate. It then executes requests in O starting from *min-s*+1 in view $v + 1$. This case arises when the primary replica is left behind other replicas.

—If *max-l* is greater than or equal to *min-s* and the request history digest $h_{max\text{-}l}$ does not match in history logs O and G, then the primary takes the same action as stated before. The primary rolls back the application and request history logs to *min-s*, inserts checkpoint certificate with sequence number *min-s* from P in its request history log, copies ordered requests from G to O starting from *min-s*+1, and then executes requests in O starting from *min-s*+1 in new view $v + 1$. This case arises when the history of the new primary diverges from the global stable state.

—If *max-l* is greater than or equal to *min-s* and the request history digest $h_{max\text{-}l}$ in O matches with that of G then requests are copied from G to O starting from *max-l*+1. The primary then executes requests starting from *max-l*+1. If the latest local stable checkpoint is less than *min-s*, then the checkpoint certificate is updated with that of the checkpoint certificate for *min-s*.

The primary also updates its local *max commit certificate* with that of *max-cc* which is computed using P as described in the protocol step VC3. It then enters view $v + 1$ and starts accepting messages in view $v + 1$.

Backup. A backup accepts a new-view message for view $v + 1$ if it is properly signed, if the view change messages it contains are valid, and if the request history log G that the primary computed is correct; the backup verifies G by performing a computation similar to the one the primary performed to create G, as described in the protocol step VC3. It then reconciles its local state using the same computation used by the primary as described earlier.

3.5.4 *The Cost of Speculation.* While Zyzzyva uses speculation to improve performance when the primary is correct, a faulty primary can impose significant overhead by wasting work performed by correct replicas before the view change. A faulty primary can send different request orders to replicas and make their state diverge. Although Zyzzyva's view change protocol ensures correctness under such an attack by rolling the system back to a consistent state, it cannot prevent a faulty primary from wasting work and slowing down the system. In the worst case, replicas lose at most $2 \times$ *CP_INTERVAL* requests worth of work as they do not execute more than $2 \times$ *CP_INTERVAL* requests speculatively since their last stable checkpoint.

3.6 Correctness

This section sketches the proof that Zyzzyva maintains properties SAF and LIV defined previously; full proofs can be found in Kotla's [2008] dissertation.

3.6.1 *Safety.* We first show that Zyzzyva's agreement subprotocol is safe within a single view and then show that the agreement and view change protocols together ensure safety across views.

—*Within a View.* The proof proceeds in two parts. First we show that no two requests complete with the same sequence number n. Then we show that for any two requests r and r' that complete with sequence numbers n and n' respectively, if $n < n'$ then h_n is a prefix of $h_{n'}$.

Part 1. A request completes when the client receives $3f + 1$ matching SPEC-RESPONSE messages in phase 1 or $2f + 1$ matching LOCAL-COMMIT messages in phase 2. If a request completes in phase 1 with sequence number n, then no other request can complete with sequence number n because correct replicas: (a) send only one speculative response for a given sequence number and (b) send a LOCAL-COMMIT message only after seeing $2f + 1$ matching SPEC-RESPONSE messages. Similarly, if a request completes with sequence number n in phase 2, no other request can complete since correct replicas only send one LOCAL-COMMIT message for sequence number n.

Part 2. For any two requests r and r' that complete with sequence numbers n and n' and histories h_n and $h_{n'}$ respectively, there are at least $2f + 1$ replicas that ordered each request. Because there are only $3f + 1$ replicas in total, at least one correct replica ordered both r and r'. If $n < n'$, it follows that h_n is a prefix of $h_{n'}$.

—*Across Views*. We show that any request that completes based on responses sent in view $v < v'$ is contained in the history specified by the NEW-VIEW message for view v'. Recall that requests complete either when a correct client receives $3f + 1$ matching speculative responses or $2f + 1$ matching LOCAL-COMMIT messages.

If a request r completes with $2f + 1$ matching LOCAL-COMMIT messages, then at least $f + 1$ correct replicas have received a commit certificate for r (or for a subsequent request) and will send that commit certificate to the new primary in their VIEW-CHANGE message. Because there are $3f + 1$ replicas in the system and $2f + 1$ VIEW-CHANGE messages in a NEW-VIEW message, that commit certificate will necessarily be included in the NEW-VIEW message and r will be included in the history. Consider instead a request r that completes with $3f + 1$ matching SPEC-RESPONSE messages and does not complete with $2f + 1$ matching LOCAL-COMMIT messages. Every correct replica will include the ORDER-REQ for r in its VIEW-CHANGE message, ensuring that the request will be supported by at least $f + 1$ replicas in the set of $2f + 1$ VIEW-CHANGE messages collected by the primary of view v' and therefore be part of the NEW-VIEW message.

3.6.2 *Liveness.* Zyzzyva guarantees liveness only during periods of synchrony. To show that a request issued by a correct client eventually completes, we first show that if the primary is correct when a correct client issues the request, then the request completes. We then show that if a request from a correct client does not complete during the current view, then a view change occurs.

Part 1. If the client and primary are correct, then protocol steps 1–3 ensure that the client receives SPEC-RESPONSE messages from all correct replicas. If the client receives $3f + 1$ matching SPEC-RESPONSE messages, the request completes, and so does our proof. A client that instead receives fewer than $3f + 1$ such messages will receive at least $2f + 1$ of them, since there are $3f + 1$ replicas and at most f of them are faulty. This client then sends a COMMIT message to all replicas (protocol step 4b). All correct replicas send a LOCAL-COMMIT message to the client (protocol step 4b.1), and, because there are at least $2f + 1$ correct replicas, the client's request completes in protocol step 4b.2.

Part 2. Assume the request from correct client c does not complete. By protocol step 4c, c resends the REQUEST message to all replicas when the request has not completed for a sufficiently long time. A correct replica, upon receiving the retransmitted request from c, contacts the primary for the corresponding ORDER-REQ message. Any correct replica that does not receive the ORDER-REQ message from the primary initiates the view change by sending an I-HATE-THE-PRIMARY message to all other replicas. Either at least one correct replica receives at least $f + 1$ I-HATE-THE-PRIMARY messages, or no correct replica receives at least $f + 1$ I-HATE-THE-PRIMARY messages. In the first case, the replicas commit to a view change; QED. In the second case, all correct replicas that did not receive the ORDER-REQ message from the primary receive it from another replica. After receiving an ORDER-REQ message, a correct replica sends a SPEC-RESPONSE to c. Because all correct replicas send a SPEC-RESPONSE message to c, c is guaranteed to receive at least $2f + 1$ such messages. Note that c must receive fewer than $2f + 1$ matching SPEC-RESPONSE messages: Otherwise, c would be able to form a COMMIT and complete the request, contradicting our initial assumption. If, however, c does not receive $2f + 1$ matching SPEC-RESPONSE messages, then c is able to form a POM message: c relays this message to the replicas which in turn initiate and commit to a view change, completing the proof.

4. IMPLEMENTATION OPTIMIZATIONS

Our implementation includes several optimizations to improve performance and reduce system cost.

Replacing signatures with MACs. Like previous work [Abd-El-Malek et al. 2005; Castro and Listov 2002; Cowling et al. 2006; Kotla and Dahlin 2004; Rodrigues et al. 2001; Yin et al. 2003], we replace most signatures in Zyzzyva with MACs and authenticators in order to reduce the computational overhead of cryptographic operations. The technical changes to each subprotocol required by replacing signatures (PKI) with authenticators (non-PKI) are described in Kotla's [2008] dissertation. Like PBFT [Castro and Listov 2002], we change the checkpoint protocol to wait for $2f + 1$ matching checkpoint messages instead of $f + 1$ matching messages in the protocol step CP3 (as described in Appendix A) when we replace signatures with authenticators. Finally, unlike the non-PKI view change protocol used in our earlier system [Kotla 2008], our current non-PKI view change subprotocol replaces signatures in PKI view change protocol messages (explained in Section 3.5.3) with authenticators in all messages by adding an additional *view-change-ack* phase similar to PBFT [Castro and Listov 2002] with the exception that the I-HATE-THE-PRIMARY messages continue to use digital signatures instead of authenticators.

Separating agreement from execution. We separate agreement from execution [Yin et al. 2003] by requiring only $2f + 1$ replicas to be execution replicas. The remaining replicas serve as witness replicas [Liskov et al. 1991], aiding in the process of ordering requests but not replicating the application. Witness replicas include *Null* as the application reply in their responses. Clients accept a history based on the agreement protocol described in the previous section with a slight modification: A pair of responses are considered to match if: (1) all the fields (v, n, h_n, c, t, OR) except the reply r and response hash $H(r)$ fields match and (2) either fields r and $H(r)$ match or one of the responses have a *Null* reply. A client acts on a reply only after receiving the appropriate number of matching responses with at least $f+1$ matching application replies from execution replicas. We can gain further benefit by biasing the primary selection criteria so that witness replicas are chosen as the primary more frequently than execution replicas. Because the primary is the bottleneck node in the system and a witness replica is under less application-level load than a regular replica, this bias can result in faster ordering and processing of requests. However, we cannot use this bias if resolving nondeterminism at the primary requires application state.

Request batching. We batch concurrent requests to reduce cryptographic and communication overheads like other agreement-based replicated services [Castro and Listov 2002; Kotla and Dahlin 2004; Rodrigues et al. 2001; Yin et al. 2003]. Batching requests amortizes the cost of replica operations across multiple requests and reduces the total number of operations per request. One key step in batching requests is having replicas compute a single history digest corresponding to the entries in the batch. This batch history is used in responses to all requests included in the batch. If the second phase completes for any request in the batch, the second phase is considered complete for all requests in the batch and replicas respond to the retransmission of any requests in the batch with LOCAL-COMMIT messages.

Caching out-of-order requests. The protocol described in Section 3.2 dictates that replicas discard ORDER-REQ messages that are received out of order. We improve performance when the network delivers messages out of order by caching these requests until the appropriate sequence number is reached. Similarly, the view change subprotocol can order additional requests that are not supported by $f+1$ speculative responses.

Read-only optimization. Like PBFT [Castro and Listov 2002], we improve the performance of read-only requests that do not modify the system state. A client sends read-only requests directly to the replicas which execute the requests immediately, without recording the request in the request history. As in PBFT, clients wait for $2f+1$ matching replies in order to complete read-only operations. In order for this optimization to function, we augment replies to read requests with a replica's max_n and *max-cc*. A client that receives $2f+1$ matching responses, including the max_n and *max-cc* fields, such that $max_n =$ *max-cc* can accept the reply to the read. Furthermore, a client that receives $3f+1$ matching replies can accept the reply to the read, even if the *max-cc* and max_n values are not consistent.

Single execution response. The client randomly selects a single execution replica to respond with a full response while the other execution replicas send only a digest of the response. This optimization is introduced in PBFT [Castro and Listov 2002] and saves network bandwidth proportional to the size of responses.

Preferred quorums. Q/U [Abd-El-Malek et al. 2005] and HQ [Cowling et al. 2006] leverage preferred quorums to reduce the cost of authenticators by optimistically including MACs for a subset of replicas rather than all replicas. With preferred quorums optimization, replicas authenticate speculative response messages for the client and a subset of $2f$ other replicas. Additionally, on the initial transmission, the client can specify that replicas should authenticate speculative response messages to the client only. This optimization reduces the number of cryptographic operations performed by backup replicas from $2+\frac{3f+1}{b}$ to $2+\frac{1}{b}$. However, the preferred quorum optimization yields no reduction in the cryptographic overhead of $2+\frac{3f}{b}$ MAC operations per request at the primary. Hence, the preferred quorum optimization provides only marginal improvement in overall application throughput.

Other optimizations. First, we use an adaptive commit timer at the client that adapts to the slowest replica in the system to initiate the commit phase. In Zyzzyva, a correct client can waste work by pessimistically initiating the commit phase as soon as it receives $2f+1$ matching speculative responses in the first phase, although none of the replicas is faulty and their responses are delayed. On the other hand, a correct client cannot wait indefinitely to receive $3f+1$ matching speculative responses as it may never hear from a faulty replica. How long should a correct client wait before initiating the commit phase? We address this dilemma by using an adaptive commit timer. We pessimistically initialize this timer to zero, which means that the client will start the commit phase as soon as it receives $2f+1$ matching responses in the first phase. If it receives the remaining of f matching speculative responses before the end of the commit phase, then the client completes the operation and also sets the commit timer to the difference between the time when the commit phase started and the time it received the $3f+1$-th matching speculative response. So, the next time around the client will start the commit timer after receiving $2f+1$ matching speculative responses but will not initiate the commit phase until the commit timer expires. If an operation ever completes at the end of the commit phase, then it resets the commit timer to zero. This mechanism adaptively and opportunistically delays the commit phase in order to avoid the unnecessary overhead of the commit phase when there are no failures while ensuring that the system is live in the presence of failures. Second, like PBFT [Castro 2001], clients broadcast requests directly to all the replicas, whereas the primary uses just the request digest in the order request message.

4.1 Making the Faulty Case Fast

Zyzzyva uses speculation to optimize performance for the common case when there are no failures in the system but also aims to improve performance for other cases when the backup nodes fail or respond slowly.

Commit optimization. In the presence of backup failures, the protocol described in Section 3.2 requires that clients start the second phase if they receive fewer than $3f + 1$ responses. Replicas then verify the commit certificate and send the local-commit response. The problem with this approach is that the replicas end up splitting the batch of requests in the first phase when replies are sent back to the clients and then verify commit messages from each client separately in the second phase. Thus, replicas fail to amortize the verification cost in the second phase of the protocol, and this increases the protocol overhead of the replicas from $2 + \frac{3f+1}{b}$ to $3 + \frac{5f+1}{b}$ MAC operations per request.

Zyzzyva addresses this problem by letting a client set a commit optimization bit in its requests as a hint to the replicas to send speculative replies only after committing the request locally. When this bit is set, backup replicas broadcast a signed ORDER-REQ message (similar to the prepare message in PBFT) to other replicas after they receive a valid ORDER-REQ request message from the primary. If a replica receives $2f + 1$ matching ORDER-REQ messages (including its own) it then commits the request locally, executes the request, and sends the speculative response to the client with both max_n and *max-cc* set to the sequence number of the request. Like read-only optimization, clients consider a request to be complete if they receive $2f + 1$ matching speculative responses with $max_n =$ *max-cc* and deliver response to the application.

A client sets the commit optimization bit in a request if it failed to complete the previous request on the fast path. The client resets this bit to zero if and when it receives $3f + 1$ matching speculative responses. This bit is initialized to zero assuming that they are no faults in the system. This optimization reduces the cryptographic overhead at a replica from $3+\frac{5f+1}{b}$ to $2+\frac{5f+1}{b}$ MAC operations per request because it allows replicas to verify commit messages once for an entire batch before committing requests locally. We evaluate the performance impact of this optimization in Section 5.4 and show that Zyzzyva performs well even in the presence of backup replica failures.

Zyzzyva5. We introduce a second protocol, Zyzzyva5 [Kotla 2008], that uses $2f$ additional *witness replicas* (the number of execution replicas is unchanged at $2f + 1$) for a total of $5f + 1$ replicas. Increasing the number of replicas lets clients receive responses in three one-way message delays even when f backup replicas are faulty [Dutta et al. 2005; Lamport 2003; Martin and Alvisi 2006]. Zyzzyva5 trades the number of replicas in the system against performance in the presence of faults. Zyzzyva5 is identical to Zyzzyva with a simple modification: Nodes wait for an additional f messages, so that, for instance, if a node bases a decision on a set of $2f + 1$ messages in Zyzzyva, the corresponding decision in Zyzzyva5 is based on a set of $3f + 1$ messages. The exceptions to this rule are the "I hate the primary" phase of the view change protocol and the fill-hole and confirm-request subprotocols that serve to prove that another correct replica has taken an action; these phases still require only $f + 1$ responses.

5. EVALUATION

This section examines the performance characteristics of Zyzzyva and compares it with existing approaches. We run our experiments on 3.0 GHz Pentium-4 machines running the Linux 2.6 kernel. We use MD5 for MACs and AdHash [Bellare and Micciancio 1997] for incremental hashing. MD5 is known to be vulnerable, but we use it to make our results comparable with those in the literature. Since Zyzzyva uses fewer MACs per request than any of the competing algorithms, our advantages over other algorithms would be increased if all were to use the more secure, but more expensive, SHA-256.

For comparison, we run Castro and Liskov's [2002] implementation of PBFT and Cowling et al.'s [2006] implementation of HQ; we scale up measured throughput for the small request/response benchmark by 9% [OpenSSL 2007] to account for their use of SHA-1 rather than MD5. We include published throughput measurements for Q/U [Abd-El-Malek et al. 2005]; we scale reported performance up by 7.5% to account for our use of 3.0 GHz rather than 2.8 GHz machines. We also compare against measurements of an unreplicated server.

Unless noted otherwise, in our experiments Zyzzyva uses all of the optimizations described in Section 4 other than preferred quorums. PBFT [Castro and Listov 2002] does not implement preferred quorum optimization. We run with preferred quorum optimization for HQ [Cowling et al. 2006]. We do not use the read-only optimization for Zyzzyva and PBFT unless we state so explicitly.

Our measured PBFT throughput of 71K ops/second on 3 GHz machines (as shown in Section 5) matches the published peak throughput numbers (15K ops/second on 600 MHz machine) [Castro and Liskov 2000] after factoring in the increased CPU speed. However, the numbers reported for PBFT in Q/U [Abd-El-Malek et al. 2005] and HQ [Cowling et al. 2006] are quite lower and do not match with ours or the numbers reported in the PBFT paper [Castro and Liskov 2000].

5.1 Throughput

To stress-test Zyzzyva we use the microbenchmarks devised by Castro and Liskov [2002]. In the 0/0 benchmark, a client sends a null request and receives a null reply. In the 4/0 benchmark, a client sends a 4KB request and receives a null reply. In the 0/4 benchmark, a client sends a null request and receives a 4KB reply.

Figure 4 shows the throughput achieved for the 0/0 benchmark by Zyzzyva, Zyzzyva5, PBFT, and HQ (scaled as noted before). For reference, we also show the peak throughput reported for Q/U [Abd-El-Malek et al. 2005] in the $f = 1$ configuration, scaled to our environment as described earlier. As the number of clients increases, Zyzzyva and Zyzzyva5 scale better than PBFT with and without batching. Without batching, Zyzzyva achieves a peak throughput that is 2.7 times higher than PBFT because of PBFT's higher cryptographic overhead (PBFT performs about 2.2 times more operations than Zyzzyva) and message overhead (PBFT sends and receives about 3.7 times more messages than Zyzzyva). When the batch size is increased to 10, Zyzzyva's and Zyzzyva5's peak throughputs increase to 86K ops/second, suggesting that the protocol overhead at the primary is 12μs per batched request. With the batch size of 10, PBFT's peak throughput increases to 59K ops/second. The 45% difference in peak throughput between Zyzzyva and PBFT is largely accounted for by PBFT's higher cryptographic overhead (about 30%) and message overhead (about 30%)

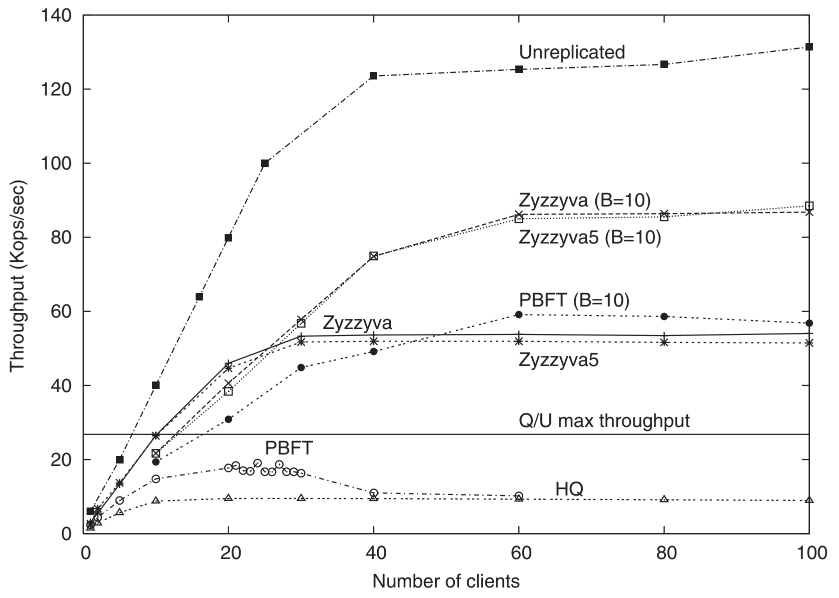


Fig. 4. Realized throughput for the 0/0 benchmark as the number of client varies for systems configured to tolerate $f = 1$ faults.

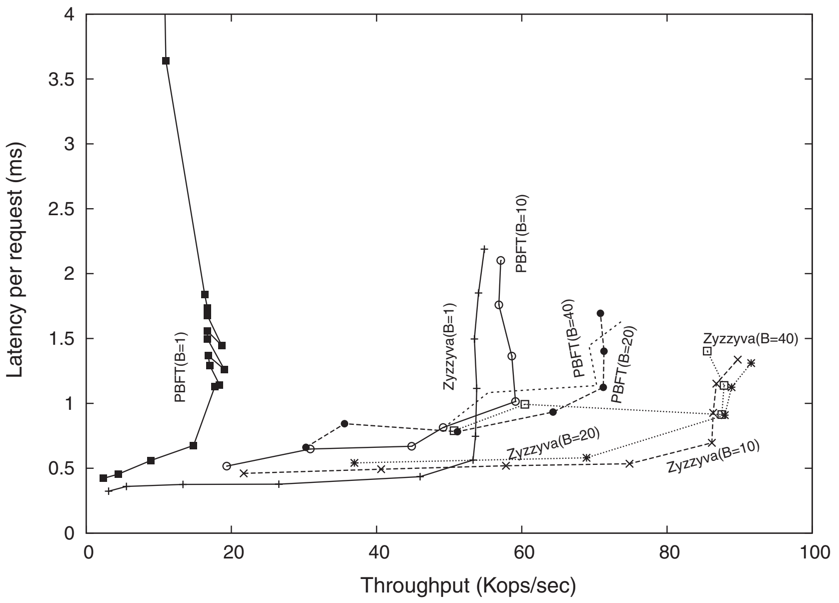


Fig. 5. Latency vs. throughput for systems with increasing batch sizes.

compared to Zyzzyva. However, as Figure 5 shows, further increases in batch size do not significantly improve Zyzzyva's performance. Conversely, PBFT's performance peaks with a batch size of 20, where Zyzzyva's throughput advantage reduces to 23%.

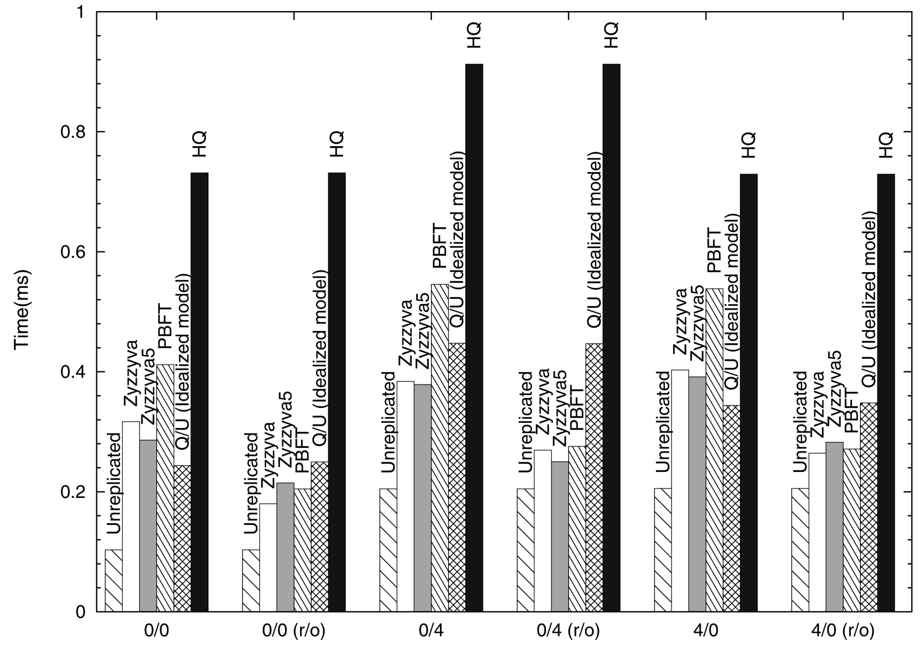


Fig. 6. Latency for 0/0, 0/4, and 4/0 benchmarks for systems configured to tolerate $f = 1$ faults.

Zyzzyva provides over 3 times the reported peak throughput of Q/U and over 9 times the measured throughput of HQ. This difference stems from three sources. First, Zyzzyva requires fewer cryptographic operations per request compared to HQ and Q/U. Second, neither Q/U nor HQ is able to use batching to reduce cryptographic and message overheads. Third, Q/U and HQ do not take advantage of the Ethernet broadcast channel to speed up the one-to-all communication steps.

Overall, the peak throughput achieved by Zyzzyva is within 35% of that of an unreplicated server that simply replies to client requests over an authenticated channel. Note that as application-level request processing increases, the protocol overhead will fall.

5.2 Latency

Figure 6 shows the latencies of Zyzzyva, Zyzzyva5, Q/U, and PBFT for the 0/0, 0/4, and 4/0 microbenchmarks. For Q/U, which can complete in fewer message delays than Zyzzyva during contention-free periods, we use a simple best-case implementation of Q/U with preferred quorums in which a client simply generates and sends $4f + 1$ MACs with a request, each replica verifies $4f + 1$ MACs (1 to authenticate the client and $4f$ to validate the object history set state), each replica generates and sends $4f + 1$ MACs (1 to authenticate the reply to the client and $4f$ to authenticate object history set state) with a reply to the client, and the client verifies $4f + 1$ MACs. We examine both the default read/write requests that use the full protocol and read-only requests that exploit the read-only optimization.

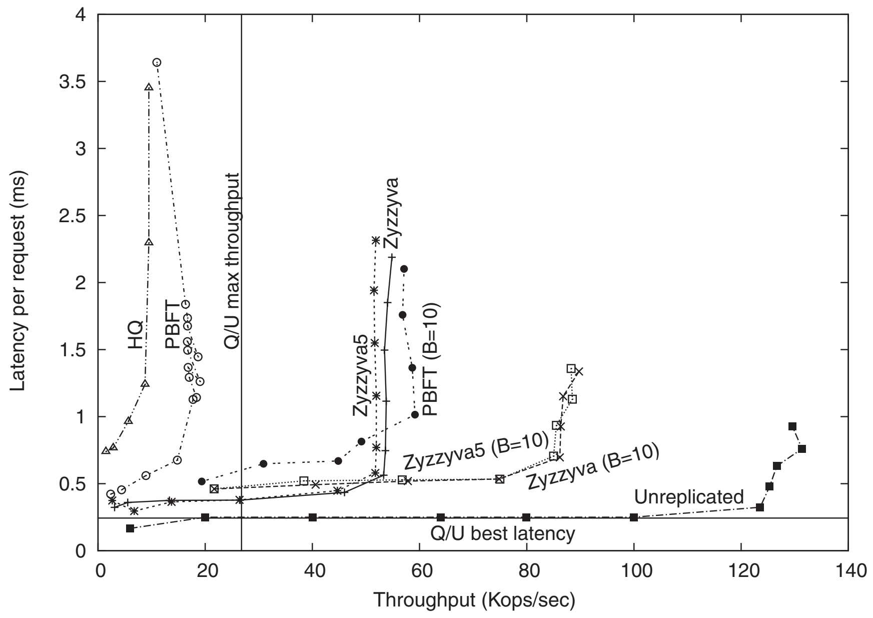


Fig. 7. Latency vs. throughput for systems configured to tolerate $f = 1$ faults.

Zyzzyva uses fast agreement to drive its latency near the optimal for an agreement protocol: three one-way message delays [Dutta et al. 2005; Lamport 2003; Martin and Alvisi 2006]. The experimental results in Figure 6 show that Zyzzyva and Zyzzyva5 achieve lower latencies than PBFT for write operations. For reads, Zyzzyva, Zyzzyva5, and PBFT are comparable. HQ performs significantly worse than Zyzzyva and PBFT because it uses unicast instead of multicast for exchanging messages, SHA1 instead of MD5 for computing message digests, and TCP instead of UDP as the transport layer. As expected, by avoiding serialization Q/U achieves even better latency in low-contention workloads such as the one examined here, though Zyzzyva and PBFT can match Q/U for read-only requests where all of these protocols can complete in two message delays.

Figure 7 shows latency and throughput as we vary offered load. As the figure illustrates, batching in Zyzzyva, Zyzzyva5, and PBFT increases latency but also increases peak throughput. Adaptively setting the batch size in response to workload characteristics is an avenue for future work.

5.3 Fault Scalability

In this section we examine how the performance of these protocols depends on the number of tolerated faults (f).

Figure 8 shows the peak throughputs of Zyzzyva, PBFT, HQ, and Q/U (reported throughput) with increasing number of tolerated faults for batch sizes of 1 and 10. Zyzzyva is robust to increasing value of f and continues to provide significantly higher throughput than the other systems. Additionally, as expected for the case with no batching, the overhead of Zyzzyva increases more

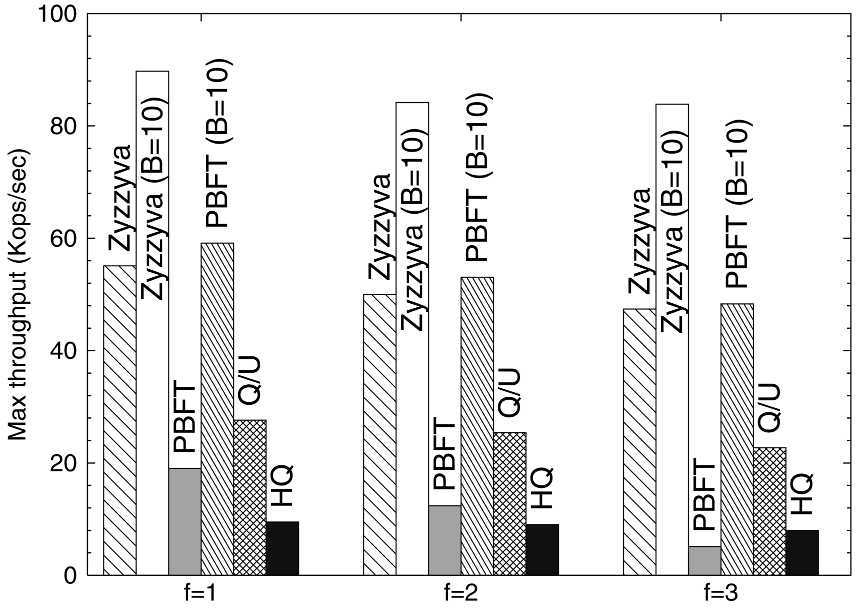


Fig. 8. Fault scalability: Peak throughputs.

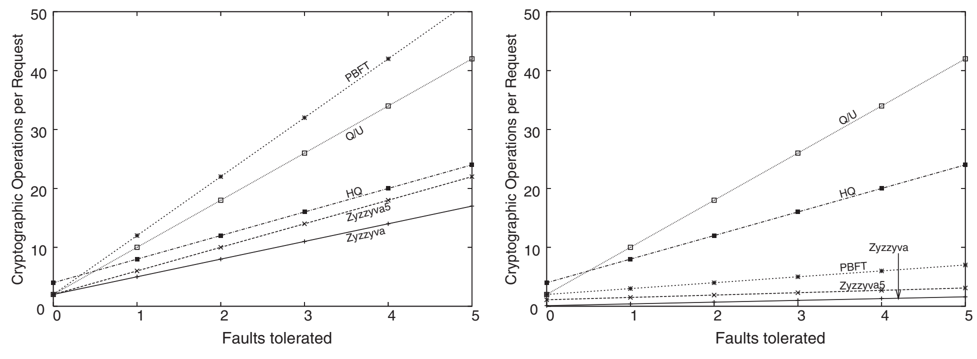
slowly than PBFT with increasing f because Zyzzyva requires $2 + (3f + 1)$ cryptographic operations compared to $2 + (10f + 1)$ cryptographic operations for PBFT.

Figure 9 shows the number of cryptographic operations per request and the number of messages sent and received per request at the bottleneck server (the primary in Zyzzyva, Zyzzyva5, and PBFT; any server in Q/U and HQ) using an analytical cost model and assuming that all protocols implement preferred quorum optimization.

Figure 9(a) shows that Zyzzyva and Zyzzyva5 scale well compared to other protocols with increasing f. Protocols that support batching, such as Zyzzyva, Zyzzyva5, and PBFT, scale even better when the batch size increases to 10 requests because they perform fewer cryptographic operations per request.

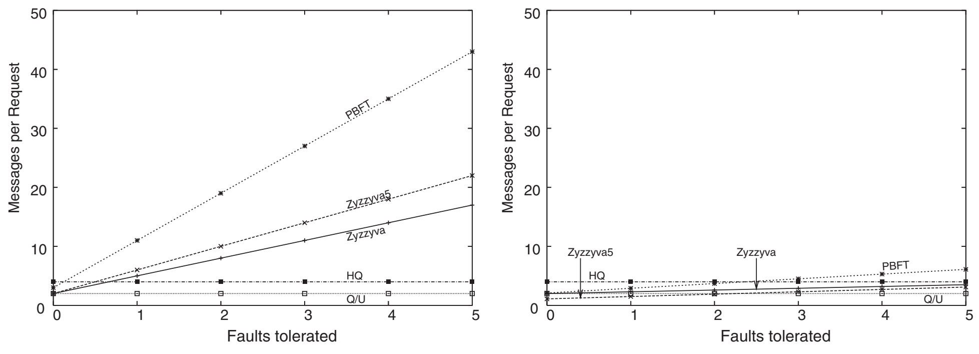
If multicast is supported, the number of messages processed by a bottlenecked server for a client request becomes approximately the same for all protocols. Multicast reduces the number of client messages for all protocols by allowing clients to transmit their requests to all servers in a single send. Multicast also reduces the number of server messages for Zyzzyva, Zyzzyva5, PBFT, and HQ (but not Q/U) when the primary or other servers communicate with their peers. In particular, with multicast the Zyzzyva primary sends or receives one message per batch of operations plus two additional messages per request, regardless of f.

Figure 9(b) plots overhead in terms of number of messages for the case when there is no multicast support. One point worth noting is that message counts at the primary for Zyzzyva, Zyzzyva5, and PBFT increase as f increases, while server message counts are constant with f for Q/U and HQ. However, with



Batch size = 1 Batch size = 10

(a) bottleneck server cryptographic operations



Batch size = 1 Batch size = 10

(b) bottleneck server messages (without multicast support)

Fig. 9. Fault scalability using the analytical model.

increasing batch sizes the message count overhead is amortized across the batch of requests and the rate of increase reduces with increasing f for Zyzzyva, Zyzzyva5, and PBFT: when the batch size is increased to 10, Zyzzyva, Zyzzyva5, and PBFT are comparable to quorum-based protocols.

Kotla [2008] examines other metrics, such as message and cryptographic overheads at the client and finds that Zyzzyva outperforms all protocols except PBFT by these metrics.

5.4 Performance During Failures

Zyzzyva guarantees correct execution with any number of faulty clients and up to f faulty replicas. However, its performance is optimized for failure-free operation, and a single faulty replica can force Zyzzyva to execute the slower two-phase protocol.

One solution is to buttress Zyzzyva's fast 1-phase path by employing additional servers. Zyzzyva5 uses a total of $5f + 1$ servers ($2f + 1$ full replicas and $3f$ additional witnesses) to allow the system to complete requests via the fast

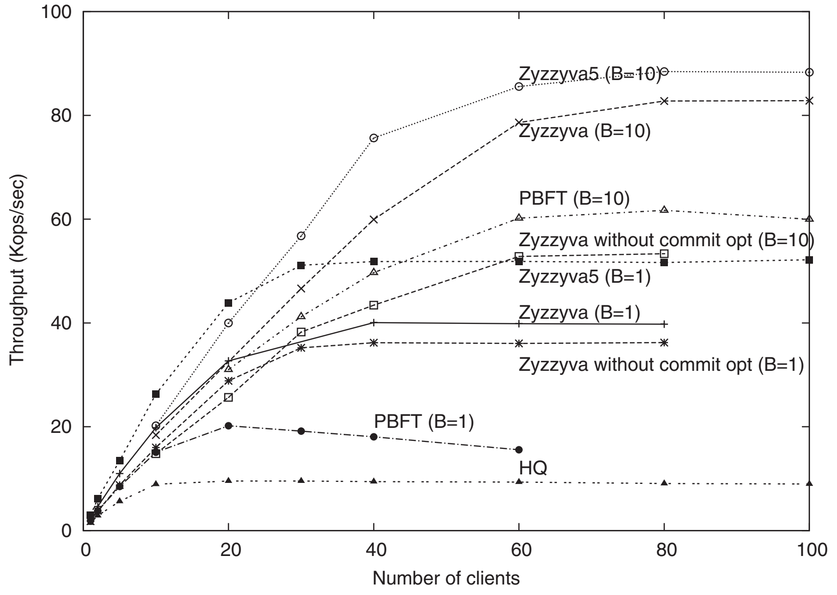


Fig. 10. Realized throughput for the 0/0 benchmark as the number of clients varies when $f = 1$ nonprimary replicas fail to respond to requests.

communication pattern shown in Figure 1(a) when the client receives $4f + 1$ (out of $5f + 1$) matching replies.

Zyzzyva remains competitive with existing protocols, even when, running with $3f + 1$ replicas, it falls back to the slower two-phase protocol. This is surprising, because Zyzzyva's cryptographic overhead at the bottleneck replica should increase from $2 + \frac{3f+1}{b}$ to $3 + \frac{5f+1}{b}$ operations per request if we simply execute the two-phase algorithm described before. (As noted at the start of this section, we omit the preferred quorums optimization in our experimental evaluations, so the $2 + \frac{3f+1}{b}$ MAC operations per request in our measurements are a higher figure than the $2 + \frac{3f}{b}$ listed in Table I.) However, as explained in Section 4.1, our implementation includes a *commit optimization* that reduces cryptographic overheads to $2 + \frac{5f+1}{b}$ cryptographic operations per request (from $3 + \frac{5f+1}{b}$) by having replicas initiate and complete the second phase to commit the request before they execute the request and send the response (with the committed history) back to the client.

Figure 10 compares the throughputs of Zyzzyva, Zyzzyva5, PBFT, and HQ in the presence of f nonprimary-server fail-stop failures. We do not include a discussion of Q/U in this section as the throughput numbers of Q/U with failures are not reported [Abd-El-Malek et al. 2005], but we would not expect a fail-stop failure by a replica to reduce significantly the performance shown for Q/U in Figure 4. Also, we do not include a line for the unreplicated server case as the throughput falls to zero when the only server suffers a fail-stop failure.

As Figure 10 shows, without the commit optimization, falling back on the two-phase protocol reduces Zyzzyva's maximum throughput from 86K ops/second

(Figure 4) to 52K ops/second. Despite this extra overhead, Zyzzyva's "slow case" performance remains within 13% of the PBFT's performance, which is less aggressively for the failure-free case and which suffers no slowdown in this scenario. Zyzzyva's commit optimization repairs most of the damage caused by a fail-stop replica, maintaining a throughput of 82K ops/second which is within 5% of the peak throughput achieved for the failure-free case. For systems that can afford extra witness replicas, Zyzzyva5's throughput is not significantly affected by the fail-stop failure of a replica, as expected. HQ continues to be outperformed for the same reasons explained in Section 5.1.

6. RELATED WORK

Zyzzyva stands on the shoulders of recent efforts that have dramatically cut the costs and improved the practicality of BFT replication. Castro and Liskov's [2002] Practical Byzantine Fault Tolerance (PBFT) protocol devised techniques to eliminate expensive signatures and potentially fragile timing assumptions, and it demonstrated high throughputs of over ten thousand requests per second. This surprising result jump started an arms race in which researchers reduced replication costs [Yin et al. 2003], and improved performance [Abd-El-Malek et al. 2005; Cowling et al. 2006; Kotla and Dahlin 2004] of BFT service replication. Zyzzyva incorporates many of the ideas developed in these protocols and folds in the new idea of speculative execution to construct an optimized fast path that significantly outperforms existing protocols and that has replication cost, processing overhead, and latency that approach the theoretical minima for these metrics. An article describing an earlier version of the Zyzzyva system appeared before [Kotla et al. 2007a].

Numerous BFT agreement protocols [Castro and Listov 2002; Cowling et al. 2006; Kotla and Dahlin 2004; Martin and Alvisi 2006; Rodrigues et al. 2001; Yin et al. 2003] have used *tentative execution* to reduce the latency experienced by clients. This optimization allows replicas to execute a request tentatively as soon as they have collected the Zyzzyva equivalent of a commit certificate for that request. This optimization may superficially appear similar to Zyzzyva's support for *speculative executions*, but there are two fundamental differences. First, Zyzzyva's speculative execution allows requests to complete at a client after a single phase, without the need to compute a commit certificate: this reduction in latency is not possible with traditional tentative executions. Second, and more importantly, in traditional BFT systems a replica can execute a request tentatively only after the replica's "state reflects the execution of all requests with lower sequence number, and these requests are all known to be committed" [Castro and Liskov 1999]. In Zyzzyva, replicas continue to execute request speculatively, without waiting to know that requests with lower sequence numbers have completed; this difference is what lets Zyzzyva leverage speculation to achieve not just lower latency but also higher throughput. In Q/U [Abd-El-Malek et al. 2005], replicas speculatively execute requests from clients without ordering them first. Thus, in the presence of request contentions, correct replicas in Q/U can misspeculate and diverge by executing contentious requests (modifying a common object or state variable) in different orders even when there are no failures in the system. In contrast, Zyzzyva's replicas speculate on the primary being correct rather than the workload, and thus do not misspeculate on request contentions when there are no failures in the system.

Q/U [Abd-El-Malek et al. 2005] provides high throughput assuming low concurrency in the system, but requires higher number of replicas than Zyzzyva. HQ [Cowling et al. 2006] uses fewer replicas than Q/U but uses multiple rounds to complete an operation. Both HQ and Q/U do not batch concurrent requests and incur higher overhead in the presence of request contention. Singh et al. [2008] compare the performance of Zyzzyva, PBFT, and Q/U under extreme network conditions and with varying message size using their simulation tool. For peak throughput, they validate our results and show that Zyzyva outperforms BFT and Q/U protocols even when the timeouts are misconfigured. However, they show that the throughput benefits of Zyzzyva over other protocols reduce with increasing request sizes and become indistinguishable for large requests, as expected, because nodes are bottlenecked by network and per-byte message processing overheads rather than by the protocol-specific cryptographic overheads. For latency under low load, they validate our results and show that Q/U's latency is comparable with Zyzzyva in a LAN setting or in a WAN setting (with slow and lossy links) when requests can be batched. However, they also show that Q/U can provide significantly lower latencies compared to Zyzzyva in a WAN setting and when there is little or no scope for batching because of the interreplica communication required by Zyzzyva.

To ensure correct operation, BFT systems require at least two-thirds of replicas to be working correctly. Hence, applications using BFT are not going to be available when more than one-third of total replicas are not available because of network partitions. BFT2F [Li and Mazières 2007] explores how to weaken gracefully the consistency guarantees provided by BFT state machine replication when the number of faulty replicas exceeds one-third but is no more than two-thirds of the total replicas. Zeno [Singh et al. 2009] requires as few as one-third of replicas to make progress but offers weaker eventual consistency semantics. Attested append-only memory [Chun et al. 2007] ensures correct operation even when half of the replicas are faulty under a stronger trust model where trusted hardware or software components implement A2M abstractions. Zyzzyva separates agreement from execution [Yin et al. 2003] to reduce the number of execution replicas to the minimal 2f + 1. The $2f + 1$ lower bound on BFT replication cost can be circumvented by using only $f + 1$ execution replicas in the failure-free case and activating additional replicas only upon failures. This approach is taken in ZZ [Wood et al. 2008], which uses virtual machines for fast replica activation.

Speculator [Nightingale et al. 2005] allows clients to complete operations sepculatively at the application level and perform client-level rollback. A similar approach could be used in conjunction with Zyzzyva to support clients that want to act on a reply optimistically, rather than waiting on the specified set of responses. Wester et al. [2009] demonstrate the benefits of client-side speculation in replicated state machines.

Recent work in BFT proposes techniques to improve robustness at the cost of common case performance. Aardvark [Clement et al. 2009b] eliminates *fragile optimizations* that maximize best-case performance but that can allow a faulty client or server to drive the system down expensive execution paths. UpRight [Clement et al. 2009a] borrows techniques from Zyzzyva and Aardvark to demonstrate the practicality of BFT in deployed cluster services such as the Zookeeper coordination service and the Hadoop file system.

7. CONCLUSION

By systematically exploiting speculation, Zyzzyva exhibits significant performance improvements over existing BFT protocols. The throughput overheads and latency of Zyzzyva approach the theoretical lower bounds for any BFT state machine replication protocol.

We believe that Zyzzyva demonstrates that BFT overheads should no longer be regarded as a barrier to using BFT replication, even for many highly demanding services.

APPENDIX

A. CHECKPOINT SUBPROTOCOL

The checkpoint subprotocol of Zyzzyva proceeds as follows.

CP1. When replica i receives the order request message for the *CP_INTERVAL*th request since the last checkpoint, the replica sends the speculative response to all other replicas in addition to the client.

Additional details. For efficiency, replica i does not include the body of the application reply r in the speculative response but just includes the hash $H(r)$.

CP2. Replica receives a commit certificate for the *CP_INTERVAL*th request, forms a checkpoint message, and relays the checkpoint message to all other replicas.

After receiving a commit certificate for the request and processing it as in step 5, replica i forms a $\langle \text{CHECKPOINT}, n, h, a, i\rangle_{\sigma_i}$ message and sends it to all replicas. n is the sequence number, h is the history digest, and a is the digest of the application state when every request in history h has been executed.

Additional details. Replica i can receive a commit certificate either from the client or by receiving $2f + 1$ matching speculative response messages directly from other replicas. The replica considers commit certificates gathered in either manner to be equivalent.

CP3. Replica receives $f+1$ matching checkpoint messages and considers the checkpoint stable.

After receiving $f + 1$ matching checkpoint messages, replica i considers the request stable and garbage collects any request with sequence number at most n and makes an up call into the application to garbage collect application state.

Replacing Signatures with MACs. Like PBFT, we replace digital signatures with authenticators for signing the protocol messages in the non-PKI version of the checkpoint protocol which has the same steps as before except that the replicas wait for $2f+1$ matching checkpoint messages in CP3 before considering the checkpoint stable.

ACKNOWLEDGMENTS

We thank R. Rodrigues, J. Cowling, and M. Abd-El-Malek for sharing source code for PBFT, HQ, and Q/U, respectively. We are grateful for M. Herlihy's and B. Liskov's feedback on earlier drafts of this article.

REFERENCES

ABD-EL-MALEK, M., GANGER, G., GOODSON, G., REITER, M., AND WYLIE, J. 2005. Fault-Scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 59–74.

AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. 2005. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 45–58.

AMAZON. 2008. Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

BELLARE, M. AND MICCIANCIO, D. 1997. A new paradigm for collision-free hashing: Incrementally at reduced cost. In *Proceedings of 14th Annual Eurocrypt Conference (Eurocrypt'97)*. 163–192.

CASTRO, M. 2001. Practical Byzantine fault tolerance. Ph.D. thesis, MIT, Cambridge, MA.

CASTRO, M. AND LISKOV, B. 1999. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*. 173–186.

CASTRO, M. AND LISKOV, B. 2000. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*. 273–288.

CASTRO, M. AND LISTOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4, 398–461.

CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. 2007. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev. 41*, 6, 189–204.

CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M., AND RICHE, T. 2009a. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. 270–290.

CLEMENT, A., MARCHETTI, M., WONG, E., ALVISI, L., AND DAHLIN, M. 2009b. Making Byzantine fault tolerant services tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 153–168.

COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 177–190.

DUTTA, P., GUERRAOUI, R., AND VUKOLIĆ, M. 2005. Best-Case complexity of asynchronous Byzantine consensus. Tech. rep. EPFL/IC/200499, EPFL.

DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *J. ACM*, 288–323.

FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2, 374–382.

GMAIL. 2006. Lost gmail emails and the future of Web apps. `http://it.slashdot.org` (12/29/06).

HERLIHY, M. AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3, 463–492.

HOTMAIL. 2004. Hotmail incinerates customer files. `http://news.com.com`, (6/3/04).

KEENEY, M., KOWALSKI, E., CAPPELLI, D., MOORE, A., SHIMEALL, T., AND ROGERS, S. 2005. Insider threat study: Computer system sabotage in critical infrastructure sectors. `http://www.cert.org/archive/pdf/insidercross051105.pdf`.

KOTLA, R. 2008. xbft: Byzantine fault tolerance with high performance, low cost, and aggressive fault isolation. Ph.D. thesis, The University of Texas at Austin, Austin, TX.

KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. 2007a. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. 45–58.

KOTLA, R. AND DAHLIN, M. 2004. High throughput Byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04)*. 575–584.

KOTLA, R., DAHLIN, M., AND ALVISI, L. 2007b. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*. 129–142.

LAMPORT, SHOSTAK, AND PEASE. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3, 382–401.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7, 558–565.

LAMPORT, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst. 6*, 2, 254–280.

LAMPORT, L. 2003. Lower bounds for asynchronous consensus. Lecture Notes in Computer Science, vol. 2584. Springer, 22–23.

LI, J. AND MAZIÈRES, D. 2007. Beyond one-third faulty replicas in Byzantine fault tolerant services. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*. 131–144.

LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., AND SHRIRA, L. 1991. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. 226–238.

MARTIN, J.-P. AND ALVISI, L. 2006. Fast Byzantine consensus. *IEEE Trans. Depend. Secure. Comput. 3*, 3, 202–215.

NIGHTINGALE, E., VEERARAGHAVAN, K., CHEN, P., AND FLINN, J. 2006. Rethink the sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 1–14.

NIGHTINGALE, E. B., CHEN, P., AND FLINN, J. 2005. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 191–205.

OPENSSL. 2007. OpenSSL. `http://www.openssl.org/`.

PEASE, M., SHOSTAK, R., AND LAMPORT, L. 1980. Reaching agreement in the presence of faults. *J. ACM 27*, 2.

PRABHAKARAN, V., BAIRAVASUNDARAM, L., AGRAWAL, N., ARPACI-DUSSEAU, H. G. A., AND ARPACI-DUSSEAU, R. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 206–220.

REITER, M. 1995. The Rampart toolkit for building high-integrity services. Lecture Notes in Computer Science, vol. 938. Springer, 99–110.

RODRIGUES, R., CASTRO, M., AND LISKOV, B. 2001. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. 15–28.

SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. 110–123.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4.

SINGH, A., DAS, T., MANIATIS, P., DRUSCHEL, P., AND ROSCOE, T. 2008. BFT protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. 189–204.

SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. 2009. Zeno: Eventually consistent Byzantine fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 169–184.

WESTER, B., COWLING, J., NIGHTINGALE, E. B., CHEN, P. M., FLINN, J., AND LISKOV, B. 2009. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 245–260.

Wood, T., Singh, R., Venkataramani, A., and Shenoy, P. 2008. ZZ: Cheap practical BFT using virtualization. Tech. rep. TR14-08, University of Massachusetts, Amherst, MA.

Yang, J., Sar, C., and Engler, D. 2006. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 131–146.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. 2003. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 253–267.

Received March 2009; revised June 2009; accepted September 2009