Zyzzyva: Speculative Byzantine Fault Tolerance

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong
Dept. of Computer Sciences
University of Texas at Austin
{kotla,lorenzo,dahlin,aclement,elwong}@cs.utexas.edu

ABSTRACT

We present Zyzzyva, a protocol that uses speculation to reduce the cost and simplify the design of Byzantine fault tolerant state machine replication. In Zyzzyva, replicas respond to a client's request without first running an expensive three-phase commit protocol to reach agreement on the order in which the request must be processed. Instead, they optimistically adopt the order proposed by the primary and respond immediately to the client. Replicas can thus 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.

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

Keywords

Byzantine fault tolerance, Speculative execution, Replication, Output commit

1. INTRODUCTION

Three trends make Byzantine Fault Tolerant (BFT) replication increasingly attractive for practical deployment. First, the growing value of data and and falling costs of hardware make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially provided by BFT replication. Second, mounting evidence of non-fail-stop behavior in real systems [2, 5, 6, 27, 30, 32, 36, 39, 40] suggest that BFT may yield significant benefits even without resorting to n-version programming [4, 15, 33]. Third, improvements to the state of the art in BFT replication techniques [3, 9, 10, 18, 33, 41] make BFT replication increasingly practical by narrowing the gap between BFT replication costs and costs already being paid for non-BFT replication. For example, by default, the Google file system uses 3-way replication of storage, which is roughly the cost of BFT replication for $f = 1$ failures with 4 agreement nodes and 3 execution nodes [41].

This paper presents Zyzzyva[1], a new protocol that uses *speculation* to reduce the cost and simplify the design of BFT state machine replication [19, 35]. Like traditional state machine replication protocols [9, 33, 41], a primary proposes an order on client requests to the other replicas. In Zyzzyva, unlike in traditional protocols, replicas speculatively execute requests without running an expensive agreement protocol to definitively establish the order. As a result, correct replicas' states may diverge, and replicas may send different responses to clients. Nonetheless, applications at clients observe the traditional and powerful abstraction of a replicated state machine that executes requests in a linearizable [13] order because replies carry with them sufficient history information for clients to determine if the replies and history are *stable* and guaranteed to be eventually committed. If a speculative reply and history are stable, the client uses the reply. Otherwise, the client waits until the system converges on a stable reply and history.

The challenge in Zyzzyva is ensuring that responses to correct clients become stable. Ultimately, replicas are responsible for ensuring that all requests from a correct client eventually complete, but a client waiting for a reply and history to become stable can speed the process by supplying information that will either cause the request to become stable rapidly within the current view or trigger a view change. Note that because clients do not require requests to commit but only to become stable, clients act on requests in one or two phases rather than the customary three [9, 33, 41].

Essentially, Zyzzyva rethinks the sync [28] for BFT: instead of pessimistically ensuring that replicas establish a final order on requests before communicating with a client, we move the output commit to the client. Leveraging the client in this way offers significant practical advantages. Compared

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

[1] Zyzzyva (ZIZ-uh-vuh) is the last word in the dictionary. According to dictionary.com, a zyzzyva is "any of various tropical American weevils of the genus Zyzzyva, often destructive to plants."

		PBFT	Q/U	HQ	Zyzzyva	State Machine Repl. Lower Bound
Cost	Total replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	3f+1 [31]
	Replicas with application state	**2f+1** [41]	5f+1	3f+1	**2f+1**	2f+1
Throughput	MAC ops at bottleneck server	2+(8f+1)/b	2+8f	4+4f	**2+3f/b**	2[†]
Latency	Critical path NW 1-way latencies	4	**2**	4	**3**	2/3[‡]

Table 1: Properties of state-of-the-art and optimal Byzantine fault tolerant service replication systems tolerating f faults, using MACs for authentication [9], and using a batch size of b [9]. 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 [11, 21, 25]; 2 one-way latencies is achievable if no concurrency is assumed. We include detailed discussion of this table in our extended technical report [17].

to state of the art protocols including PBFT [9, 33, 41], Q/U [3], and HQ [10], Zyzzyva reduces cryptographic overheads and increases peak throughput by a factor of two to an order of magnitude for demanding workloads. In fact, Zyzzyva's replication costs, processing overheads, and communication latencies approach their theoretical lower bounds.

1.1 Why another BFT protocol?

The state of the art for BFT state machine replication is distressingly complex. In a November 2006 paper describing Hybrid-Quorum replication (HQ replication) [10], Cowling et al. draw the following conclusions comparing three state-of-the-art protocols (Practical Byzantine Fault Tolerance (PBFT) [9, 18, 33, 41], Query/Update (Q/U) [3], and HQ replication [10]):

- "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 best choice, else HQ is the best since it outperforms [P]BFT with a batch size of 1." [10]
- "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." [10]
- "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)." [10]

Such complexity represents a barrier to 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 1 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 with respect to the total number of replicas that participate in the protocol and 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 the relative cost of application vs. agreement node replication.

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. Q/U and HQ's inability to support batching increases the work done at the bottleneck node by factors approaching 5 and 4, respectively, when one fault 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 accepted lower bound in the distributed systems literature for agreeing on a client request [11, 21, 25] and improves upon both PBFT and HQ. Q/U sidesteps this lower bound by providing a service that is slightly weaker than state machine replication (i.e., it does not put a total order on 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 "extra" latency.

With respect to fault scalability [3], the metrics that depend on f grow as slowly or more slowly in Zyzzyva as in any other protocol.

Note that as is customary [3, 9, 10, 33, 41], Table 1 compares the protocols' performance during the expected common case of fault-free, timeout-free execution. All of the protocols are guaranteed to operate correctly in the presence of up to f faults and arbitrary delays, but all of these protocols can pay significantly higher overheads and latencies in such scenarios. In §5.4, we consider the susceptibility of these protocols to faults and argue that Zyzzyva remains the most attractive choice.

2. SYSTEM MODEL

We assume the Byzantine failure model where faulty nodes (replicas or clients) may behave arbitrarily. We assume a strong adversary that can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures. In the public-key version of our protocol, we denote a message X signed by principal Y's public key as $\langle X\rangle_{\sigma_Y}$. Our system ensures its safety and liveness properties if at most f replicas are faulty. We assume a finite client population, any number of which may be faulty.

Our system's safety properties hold in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them, or deliver them out of order. Liveness, however, is ensured only during intervals in which messages sent to correct nodes are processed within some fixed (but potentially unknown) worst case delay from when they are sent.

Our system implements a BFT service using state machine replication [9, 18, 35]. Traditional state machine replication techniques can be applied only to deterministic services. We cope with the non-determinism present in many real-word applications (such as file systems [26] and databases [38]) by abstracting the observable application state at the replicas and using the agreement stage to resolve divergences [33].

Services limit the damage done by Byzantine clients by authenticating clients, enforcing access control to deny clients access to objects they do not have a right to, and (optionally) by maintaining multiple versions of shared data (e.g., snapshots in a file system [34, 16]) so that data can be recovered from older versions if a faulty client destroys data [14].

3. PROTOCOL

Zyzzyva is a state machine replication protocol based on three sub-protocols: (1) agreement, (2) view change, and (3) checkpoint. The *agreement* protocol orders requests for execution by the replicas. The *view change* protocol coordinates the election of a new primary when the current primary is faulty or the system is running slowly. The *checkpoint* protocol limits the state that must be stored by replicas and reduces the cost of performing view changes.

Principles and Challenges.

Zyzzyva focuses on safety properties *as they are observed by the client.* In Zyzzyva, replicas can become temporarily inconsistent with one another, but clients detect inconsistencies, drive replicas to converge on a single total ordering of requests, and only rely on responses that are consistent with this total order.

Given the duties BFT replication protocols already place on clients [3, 9, 10, 22, 33, 41], it is not a large step to fully move the output commit to the client, but this small step pays big dividends. First, Zyzzyva leverages speculative execution—replicas execute a request *before* its order is fully established. Second, Zyzzyva leverages fast agreement protocols [11, 21, 25] to establish a request ordering in as few as three message delays. Third, the agreement sub-protocol stops working on a request once a client knows the request's order, thereby avoiding work that would otherwise be needed to establish this knowledge at the replicas.

These choices lead to two key challenges in designing Zyzzyva. First, we must specify carefully the conditions under which a request *completes* at a client and define agreement, checkpoint, and view change sub-protocols to retain the abstraction that requests execute on a single, correct state machine. Intuitively, a request completes when a correct client may safely act on the reply to that request. To help a client determine when it is appropriate to act on a reply, Zyzzyva appends history information to the replies received by a client so that the client can judge whether the replies are based on the same ordering of requests. Zyzzyva ensures the following safety condition:

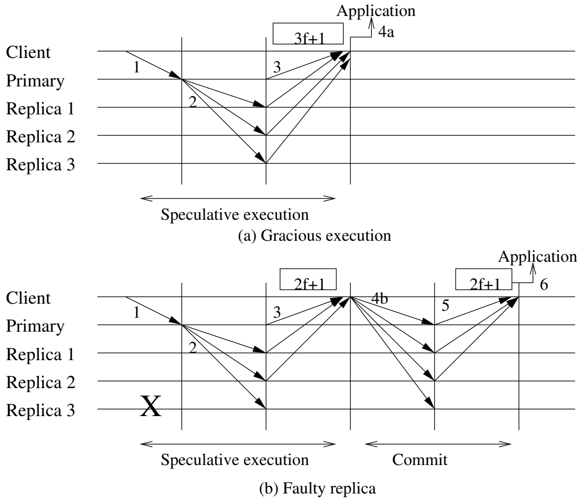


Figure 1: Protocol communication pattern within a view for (a) gracious execution and (b) faulty replica cases. The numbers refer to the main steps of the protocol numbered in the text.

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.

Second, the view change sub-protocol must ensure liveness despite an agreement sub-protocol that never requires more than two phases to complete during a view. We shift work from the agreement sub-protocol to the view change sub-protocol by introducing a new "I hate the primary" phase that guarantees that a correct replica only abandons the current view if it can ensure that all other correct replicas will join the mutiny. Zyzzyva ensures the following liveness condition under eventual synchrony[2] [12]:

LIV Any request issued by a correct client eventually completes.

Protocol Overview.

Zyzzyva is executed by $3f+1$ replicas, and execution is organized into a sequence of views. Within a view, a single replica is designated as the primary responsible for leading the agreement sub-protocol.

Figure 1 shows the communication pattern for a single instance of our client-centric fast agreement sub-protocol. A client 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 in one of two ways. First, if the client receives $3f+1$ mutually-consistent *responses* (including an application-level *reply* and the *history* on which it depends), then the client considers the request complete and acts on it. Second, if the client receives between $2f+1$ and $3f$ mutually-consistent responses, then the client gathers $2f+1$ responses and distributes this *commit certificate* to the replicas. Once $2f+1$ replicas acknowledge receiving a

[2] In practice eventual synchrony can be achieved by using exponentially increasing timeouts [9].

commit certificate, the client considers the request *complete* and acts on the corresponding reply.

If a sufficient number of replicas suspect that the current primary is faulty, then a view change occurs and a new primary is elected.

In the rest of this section, we describe the basic protocol and outline the proof of its correctness [17]. In §4 we describe a number of optimizations, all implemented in our prototype, that reduce encryption costs by replacing public key 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 by including additional witness replicas.

In §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 2(a) even when up to f non-primary replicas are faulty.

3.1 Node State and Checkpoint Protocol

To ground our discussion in definite terms, 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 (defined below) 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 committed is similar to the one used by earlier BFT protocols [9, 10, 18, 33, 41], so we defer a detailed discussion to our extended technical report [17]. However, to summarize briefly: when a correct replica generates a tentative checkpoint, it sends a signed CHECKPOINT message to all replicas. The message includes the highest sequence number of any request included in the checkpoint and a digest of the corresponding tentative checkpoint and application snapshot. A correct Zyzzyva replica considers the checkpoint and corresponding application snapshot stable when it collects $f + 1$ matching CHECKPOINT messages signed by different replicas.

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

Finally, each replica maintains a *response cache* containing a copy of the latest ordered request from, and corresponding response to, each client.

3.2 Agreement Protocol

Figure 1 illustrates the basic flow of the agreement subprotocol during a view. Because replicas execute requests speculatively in the order proposed by the primary without communicating with other replicas, the key challenge is ensuring that clients only act upon replies that correspond to stable requests executed in a total order that is guaranteed to eventually *commit* at all correct servers. The protocol is constructed so that a request *completes* at a client when the client receives $3f + 1$ matching responses or acknowledgements from $2f + 1$ replicas that they have received a *commit certificate* comprising a *local commit* from $2f + 1$ replicas. Either of these conditions serves to prove that the request will eventually be *committed* at all correct replicas with the same sequence number and history of preceding requests observed by the client.

Label	Meaning
c	Client ID
CC	Commit certificate
d	Digest of client request message $d = H(m)$
i, j	Server IDs
h_n	History through sequence number n $h_n = H(h_{n-1}, d)$
m	Message containing client request
max_n	Max sequence number accepted by replica
n	Sequence number
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

Table 2: Labels given to fields in messages.

To describe how the system deals with this and other challenging, but standard, issues—lost messages, faulty primary, faulty clients, etc.—we follow a request through the system, defining the rules a server uses to process each message. The numbers in Figure 1 correspond to numbers in the text identifying major steps in the protocol and Table 2 summarizes the labels we give fields in messages. Most readers will be happier if on their first reading they skip the text marked Additional Pedantic Details.

1. Client sends request to the primary.

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

Additional Pedantic Details: If the client guesses the wrong primary, the retransmission mechanisms discussed in step 4c below forwards the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests.

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

When the primary p receives message $m = \langle$REQUEST, o, t, $c\rangle_{\sigma_c}$ from client c, the primary assigns a sequence number n in view v to the request and relays a message $\langle\langle$ORDER-REQ, v, n, h_n, d, $ND\rangle_{\sigma_p}$, $m\rangle$ to the backup replicas where v indicates the view in which the message is being sent, n is the proposed sequence number for m, $d = H(m)$ is the digest of m, $h_n = H(h_{n-1}, d)$ is a digest summarizing the history, and

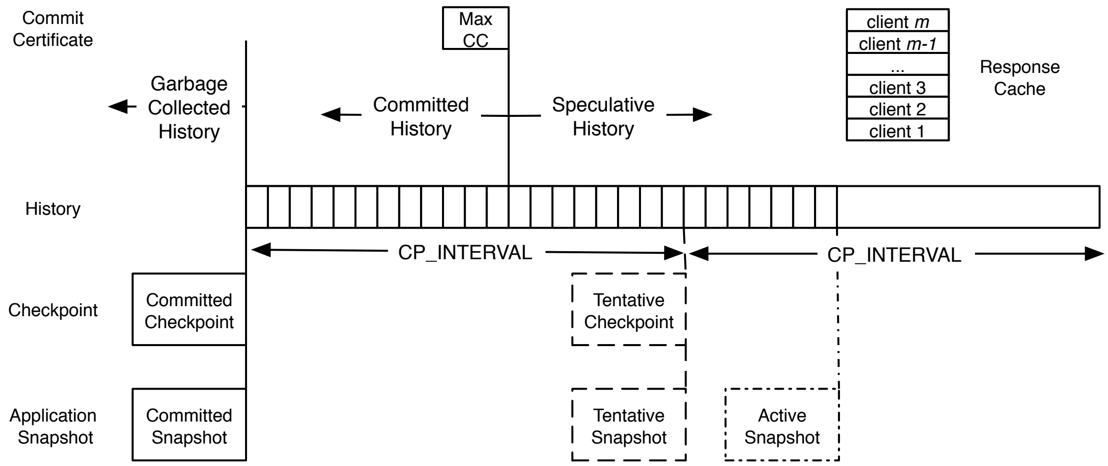


Figure 2: State maintained at each replica.

ND is a set of values for non-deterministic application variables (time in file systems, locks in databases, etc.) required for execution.

Additional Pedantic Details: The primary only takes the above 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.

Upon receipt of a message $\langle\langle$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 digest of m, $n = max_n + 1$ 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 $\langle\langle$SPEC-RESPONSE, v, n, h_n, $H(r)$, c, $t\rangle_{\sigma_i}$, i, r, $OR\rangle$ where $OR =\langle$ORDER-REQ, v, n, h_n, d, $ND\rangle_{\sigma_p}$.

Additional Pedantic 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 $\langle$FILL-HOLE, v, $max_n + 1$, n, $i\rangle_{\sigma_i}$ to the primary, and starts a timer. Upon receipt of a message $\langle$FILL-HOLE, v, k, n, $i\rangle_{\sigma_i}$ from replica i, the primary p sends a $\langle\langle$ORDER-REQ, v, n', $h_{n'}$, d, $ND\rangle_{\sigma_p}$, $m'\rangle$ to i for each request m' that p ordered in $k \leq n' \leq n$ 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 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.

4. Client gathers speculative responses.

The client receives messages $\langle\langle$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, from the replicas. SPEC-RESPONSE messages from distinct replicas *match* if they have identical v, n, h_n, $H(r)$, c, t, and r fields. There are four cases to consider. The first three handle varying numbers of matching speculative replies without considering the OR field, while the last considers only the OR field.

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

In the absence of faults, the client receives matching SPEC-RESPONSE messages from all $3f+1$ replicas. The client then considers the request and its history to be *complete* and delivers the reply r to the application. Zyzzyva guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response. Notice that although the client has a proof that the request's place in history is irrevocably set, no server has such a proof. Indeed, a server at this point cannot determine whether a request has completed in its final order or a roll-back of the server's state will be necessary because a faulty primary ordered the request inconsistently across replicas.

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

If the network, primary, or some replicas are faulty, the client c may never receive responses from all $3f + 1$ replicas. The client therefore 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 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 Pedantic 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 $H(r)$.

4b.1. 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 (1) updates its *max commit certificate* state if this certificate's sequence number is higher than the stored certificate's sequence number and (2) sends a message $\langle$LOCAL-COMMIT$, v, d, h, i, c\rangle_{\sigma_i}$ to c.

Additional Pedantic Details: If the local history simply has holes encompassed by CC's history, then i fills them as described in 3. If, however, the two histories contain different requests for the same sequence number, then i initiates the view change protocol.

4b.2. Client receives a 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 and its history to be *complete* and delivers the reply r to the application. The system guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response.

Additional Pedantic 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 step 4c described below.

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.

Client. If the network or primary is faulty, the client c may never receive matching SPEC-RESPONSE messages from $2f+1$ replicas. The client therefore sets a second timer when it first issues a request and resends the REQUEST message to all replicas when the second timer expires. It then resets its timers and continues gathering speculative responses.

Replica. When non-primary replica i receives a message $\langle$REQUEST$, o, t, c\rangle_{\sigma_c}$ from client c there are two possible actions for i to take. If the request matches or has a lower client-supplied timestamp than the currently cached request for client c, then i resends the cached response to c. If instead the request has a higher timestamp than the currently cached response, then i sends a message $\langle$CONFIRM-REQ, v, m, $i\rangle_{\sigma_i}$ where $m = \langle$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 above. If the timer fires before the primary orders the request, the replica initiates a view change.

Primary. Upon receiving the confirm request message $\langle$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 the next sequence number to order as described in step 2; otherwise, p sends to i the cached ORDER-REQ message for the most recent request from c.

Additional Pedantic Details: 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 received a commit certificate for the retransmitted request. Additionally, 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 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.

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 $OR = \langle$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, then the pair of ORDER-REQ messages constitutes a proof of misbehavior (POM) against the primary. Upon receipt of a POM, c sends a message $\langle$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.

Note that cases 4b and 4c are not exclusive of 4d; a client may receive messages sufficient to complete a request or form a commit certificate and also a proof of misbehavior against the primary.

3.3 View Changes

Fast agreement and speculative execution have profound effects on Zyzzyva's view change sub-protocol. In this section we highlight the differences between the Zyzzyva view change sub-protocol and that of previous systems. For completeness we include the full view change sub-protocol in the appendix.

The view change sub-protocol 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 sub-protocols [9, 10, 18, 33, 41] 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 sub-protocol 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 significant 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 sub-protocol. We explain the need for and details of this addition below 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 below by considering *The Case of the Uncommitted Request.*

3.3.1 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 *(pre-prepare)* 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 sub-protocols. No replicated service that uses the traditional view change protocol can be live without an agreement protocol that includes both the *prepare* and *commit* full exchanges.[3] To see how this constraint applies to Zyzzyva, 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 can 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 sub-protocol? 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 view v 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 sub-protocol 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 a 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 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 sub-protocol, which is run only when the primary is faulty.

3.3.2 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: it occurs when 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 set of $2f + 1$ 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 sub-protocol 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 or commit certificate. 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 num-

[3]Unless a client can unilaterally initiate a view change. This option is unattractive when clients can be Byzantine.

ber 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.

3.4 Correctness

This section sketches the proof that Zyzzyva maintains properties SAF and LIV defined above. Full proofs are available in the extended technical report [17].

3.4.1 Safety

We first show that our agreement sub-protocol 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. Second we show that h_n is a prefix of $h_{n'}$ for $n < n'$ and completed requests r and r'.

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-commits.

If a request r completes with $2f+1$ matching local-commits, 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.

3.4.2 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 through 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 which 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 [3, 9, 10, 18, 33, 41], we replace most signatures in Zyzzyva with MACs and authenticators in order to reduce the computational overhead of cryptographic operations. The only signatures that are not replaced with MACs are client request retransmissions and the messages of the view change protocol. The technical changes to each sub-protocol required by replacing signatures with authenticators are described in [17]. The most noticeable difference in the agreement sub-protocol is the way Zyzzyva addresses the scenario in which replica i is unable to authenticate a client request; i cannot distinguish whether the fault lies with the primary or the client. Our procedure in this case is similar to a view change and results in correct replicas agreeing to accept the request or replace it with a *no-op* in the sequence. The checkpoint sub- protocol adds a third phase to ensure that stable checkpoints are consistent with requests that complete through speculative execution. Finally, the view change sub-protocol includes an additional phase for gathering checkpoint and commit certificate proofs as is done in PBFT [9].

Separating Agreement from Execution.

We separate agreement from execution [41] by requiring only $2f+1$ replicas to be execution replicas. The remaining replicas serve as witness replicas [23], aiding in the process of ordering requests but not replicating the application. 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 even if the response r and response hash $H(r)$ fields are not identical. A client acts on a reply only after receiving the appropriate number of matching responses and $f+1$ matching application replies from execution replicas. One consequence of this optimization is that a client may have to wait until it has received more than $2f+1$ responses before it can act in the second phase. We gain further benefit by biasing the primary selection criteria so that witness replicas are chosen as the primary more frequently than execution replicas. This favoritism reduces processor contention at the primary and allows requests to be ordered and processed faster.

Request Batching.

We batch concurrent requests to reduce cryptographic and communication overheads like other agreement-based replicated services [9, 18, 33, 41, 37]. 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 request 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 sub-protocol can order additional requests that are not supported by $f+1$ speculative responses.

Read-Only Optimization.

Like PBFT [9], 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, even if the $\max_{CC}$ and $\max_n$ values are not consistent, can accept the reply to the read.

Single Execution Response.

The client specifies 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 [9] and saves network bandwidth proportional to the size of responses.

Preferred Quorums.

Q/U [3] and HQ [10] leverage preferred quorums to reduce the size of authenticators by optimistically including MACs for a subset of replicas rather than all replicas. We implement preferred quorums for the second phase; replicas authenticate speculative response messages for the client and a subset of $2f$ other replicas. Additionally, on the initial transmission, we allow the client to specify that replicas should authenticate speculative response messages to the client only. This optimization reduces the number of cryptographic operations performed by backup replicas to three while existing BFT systems [9, 18, 3, 10, 33, 41] require a linear number of cryptographic operations at each replica.

4.1 Making the Faulty Case Fast

We introduce a second protocol, Zyzzyva5, 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 message delays even when f replicas are faulty [11, 21, 25]. Zyzzyva5 trades the number of replicas in the deployed system against performance in the presence of faults. Zyzzyva5 is identical to Zyzzyva with a simple modification—nodes wait for an additional f messages, i.e. 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 sub-protocols 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 with the

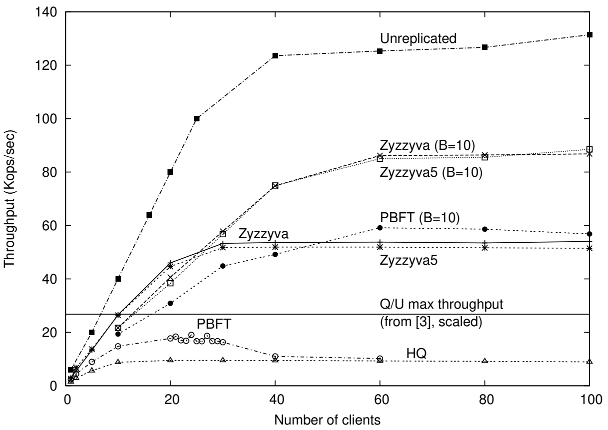


Figure 3: Realized throughput for the 0/0 benchmark as the number of client varies for systems configured to tolerate $f = 1$ faults.

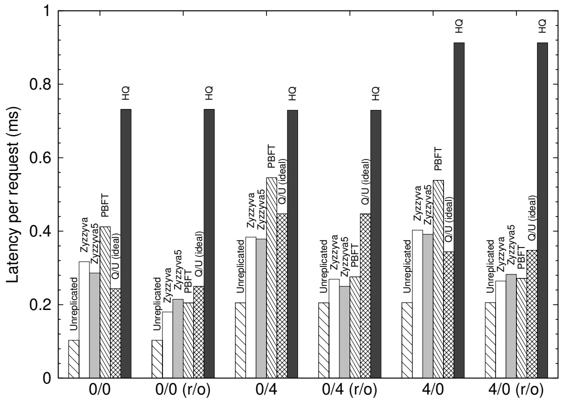


Figure 4: Latency for 0/0, 0/4, and 4/0 benchmarks for systems configured to tolerate $f = 1$ faults.

Linux 2.6 kernel. We use MD5 for MACs and AdHash [7] 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 we were to use the more secure, but more expensive, SHA-256.

For comparison, we run Castro et al.'s implementation of PBFT [9] and Cowling et al.'s implementation of HQ [10]; we scale up measured throughput for the small request/response benchmark by 9% [1] to account for their use of SHA-1 rather than MD5. We include published throughput measurements for Q/U [3]; we scale reported performance up by 7.5% to account for our use of 3.0 GHz rather than 2.8GHz machines. We also compare against measurements of an unreplicated server.

Unless noted otherwise, in our experiments we use all of the optimizations other than preferred quorums for Zyzzyva as described in §4. PBFT [9] does not implement preferred quorum optimization. We run with preferred quorum optimization for HQ [10]. We do not use the read-only optimization for Zyzzyva and PBFT unless we state so explicitly.

5.1 Throughput

To stress-test Zyzzyva we use the micro-benchmarks devised by Castro et al. [9]. 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 3 shows the throughput achieved for the 0/0 benchmark by Zyzzyva, Zyzzyva5, PBFT, and HQ (scaled as noted above). For reference, we also show the peak throughput reported for Q/U [3] in the $f = 1$ configuration, scaled to our environment as described above. 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 due to PBFT's higher cryptographic overhead (PBFT performs about 2.2 times more crypto 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/sec suggesting that the protocol overhead at the primary is 12μs per batched request. With batching, PBFT's throughput increases to 59K ops/sec. The 45% difference between Zyzzyva and PBFT's peak throughput are largely accounted for PBFT's higher cryptographic overhead (about 30%) and message overhead (about 30%) compared to Zyzzyva. 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 request over an authenticated channel. Note that as application-level request processing increases, the protocol overhead will fall.

5.2 Latency

Figure 4 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 $4f + 1$ to validate the OHS state), each replica generates and sends $4f + 1$ MACs (1 to authenticate the reply to the client and $4f$ to authenticate OHS 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.

Zyzzyva uses fast agreement to drive its latency near the optimal for an agreement protocol—3 one-way message delays [11, 21, 25]. The experimental results in Figure 4 show that Zyzzyva and Zyzzyva5 achieve significantly lower latency than the other agreement-based protocols, PBFT and HQ. As expected, Q/U's avoidance of serialization gives it even better latency in low-contention workloads such as the one examined here, though Zyzzyva and PBFT can match

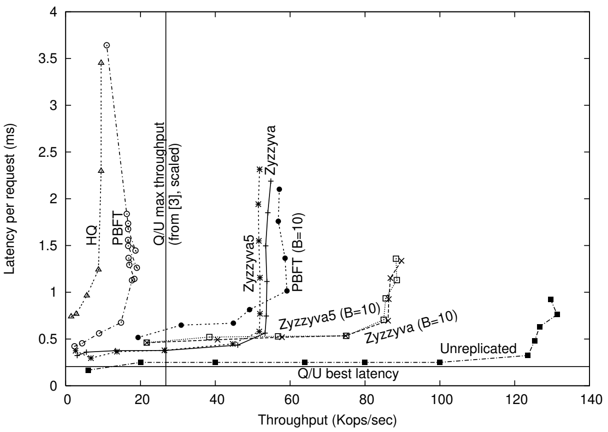


Figure 5: Latency vs. throughput for systems configured to tolerate $f = 1$ faults.

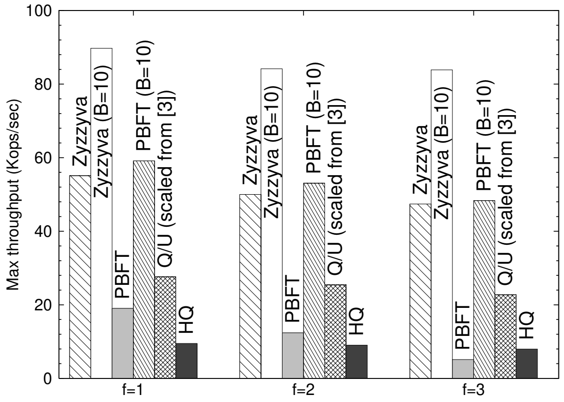


Figure 6: Fault scalability: Peak throughputs

Q/U for read-only requests where all of these protocols can complete in two message delays.

Figure 5 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 performance of these protocols as f, the number of tolerated faults, increases.

Figure 6 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 f and continues to provide significantly higher throughput than other systems for the same reasons as explained in the throughput section. Additionally, as expected for the case with no batching, the overhead of Zyzzyva increases more slowly than PBFT with increasing f because Zyzzyva requires $2 + (3f + 1)$ cryptographic operations compared to $2 + (10f + 1)$ cryptographic operations for PBFT.

Figures 7 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, PBFT, and any server in Q/U and HQ). We believe that for these metrics, the most interesting regions are

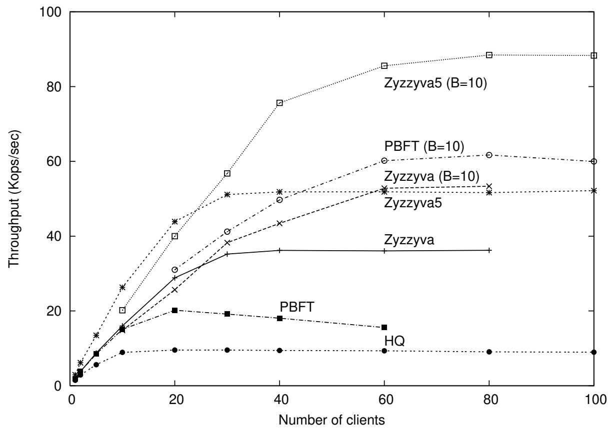


Figure 8: Realized throughput for the 0/0 benchmark as the number of client varies when f non-primary replicas fail to respond to requests.

when f is small and when batching is enabled. Not coincidentally, Zyzzyva performs well in these situations, dominating all of the approaches with respect to load at the bottleneck server. Also, when f is small, Zyzzyva and Zyzzyva5 also have low message counts at the primary.

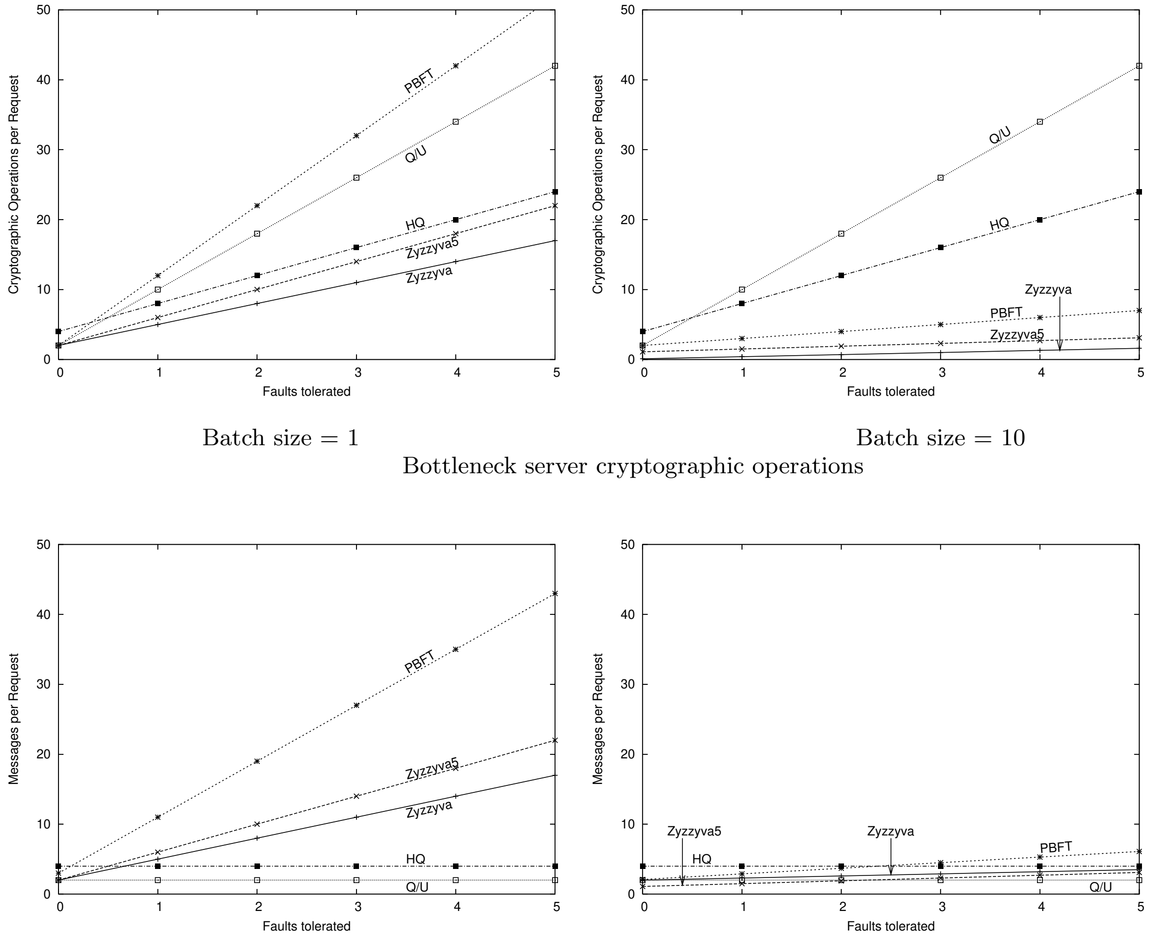
As f increases, when batching is used, Zyzzyva and Zyzzyva5 are likely to remain attractive. One point worth noting is that message counts at the primary for Zyzzyva, Zyzzyva5, and PBFT increase as f increases, while server message counts are constant with f for Q/U and HQ. In this figure, message counts do not include the multicast optimization we exploited in our experiments. Multicast reduces the number of client messages for all protocols by allowing clients to transmit their requests to all servers in one 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 primary sends or receives one message per batch of operations plus an additional two messages per request regardless of f.

The extended technical report [17] examines other metrics such as client load and finds, for example, that Zyzzyva improves upon all of the protocols except PBFT by this metric. These graphs are omitted due to space constraints.

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 the expected case of failure-free operation. In particular a single faulty replica can force Zyzzyva to execute the slower 2 phase protocol. Zyzzyva's protocol, however, remains relatively efficient in such scenarios. In particular, Zyzzyva's cryptographic overhead increases from $2 + \frac{3f+1}{b}$ to $3 + \frac{5f+1}{b}$ operations per batch. Zyzzyva5's increased fault tolerance ensures that its overheads do not increase in such scenarios, remaining at $2 + \frac{5f+1}{b}$ per batch. For comparison, PBFT uses $2 + \frac{10f+1}{b}$ operations in both this scenario and fault-free.

Figure 8 compares throughput with increasing numbers of clients for Zyzzyva, Zyzzyva5, PBFT, and HQ in the presence of f backup server failures. For the case when $f = 1$, with one failure and no batching ($b = 1$), Zyzzyva and



Batch size = 1

Batch size = 10

Bottleneck server cryptographic operations

Batch size = 1

Batch size = 10

Bottleneck server messages

Figure 7: Fault scalability using analytical model

Zyzzyva5 provide 1.8 and 2.6 times higher throughput than PBFT, respectively, because of additional cryptographic and message overheads as described above. However, with the batch size of 10, PBFT performs 15% better than Zyzzyva as the cryptographic overhead of Zyzzyva is 16% higher than PBFT. As expected, Zyzzyva5 achieves the peak throughput that is same as the peak throughput for the fault-free case and continues to outperform PBFT by 43% with batching. For the same reasons as described in the throughput section, Zyzzyva, Zyzzyva5, and PBFT outperform HQ. We do not include a discussion of Q/U in this section as the throughput numbers of Q/U with failures are not reported [3].

A limitation Zyzzyva and Zyzzyva5 share with PBFT (and HQ during periods of contention) is that a faulty primary can significantly prevent progress. These protocols replace the primary to ensure progress. Although Q/U avoids having a primary, it shares a corresponding vulnerability: a faulty client that fails to adhere to the back-off protocol can impede progress indefinitely.

6. RELATED WORK

Starting with PBFT [9, 33] several systems [3, 10, 18, 41] have explored how to make Byzantine services practical. We have discussed throughout the paper how Zyzzyva builds upon these systems and how it departs from them. As its predecessors, Zyzzyva leverages ideas inspired by Paxos [20] and by work on Byzantine quorum systems [24]. In particular, Zyzzyva fast agreement protocol is based on recent work on fast Paxos [11, 21, 25].

Numerous BFT agreement protocols [9, 10, 18, 25, 33, 41] 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" [8]. 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.

Q/U [3] provides high throughput assuming low concurrency in the system but requires higher number of replicas than Zyzzyva. HQ [10] uses fewer replicas than Q/U but uses multiple rounds to complete an operation. Both HQ and Q/U fail to batch concurrent requests and incur higher overhead in the presence of request contention; Singh et al. [37] add a preserializer to HQ and Q/U to address these issues.

BFT2F [22] explores how to gracefully weaken 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.

Speculator [29] allows clients to speculatively complete operations 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.

7. CONCLUSION

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

8. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CNS-0720649, CNS-0509338, and CNS-0411026. Ramakrishna Kotla was awarded SOSP student travel scholarship, supported by Hewlett-Packard, to present the paper at the conference. We thank Rodrigo Rodrigues, James Cowling, and Michael Abd-El-Malek for sharing source code for PBFT, HQ, and QU respectively.

9. REFERENCES

[1] OpenSSL. `http://www.openssl.org/`.

[2] US secret service report on insider attacks. `http://www.sei.cmu.edu/about/press/insider-2005.html`.

[3] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. SOSP*, Oct. 2005.

[4] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC*, pages 149–155, Nov. 1977.

[5] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE TODSC*, 1(1):87–96, 2004.

[6] L. Bassham and W. Polk. Threat assessment of malicious code and human threats. Technical report, NIST, Computer Security Division, 1994.

[7] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementally at reduced cost. In *EUROCRYPT97*, 1997.

[8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI*, February 1999.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, Nov. 2002.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, Nov. 2006.

[11] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report EPFL/IC/200499, EPFL, Feb. 2005.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[13] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Sys.*, 12(3), 1990.

[14] S. King and P. Chen. Backtracking intrusions. In *Proc. SOSP*, 2003.

[15] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *Software Engineering*, 12(1):96–109, Jan. 1986.

[16] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. In *USENIX Annual Technical Conference*, Monterey, CA, June 2007.

[17] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *University of Texas at Austin, Technical Report: UTCS-TR-07-40*, 2007.

[18] R. Kotla and M. Dahlin. High-throughput byzantine fault tolerance. In *DSN*, June 2004.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[20] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[21] L. Lamport. Lower bounds for asynchronous consensus. In *Proc. FUDICO*, pages 22–23, June 2003.

[22] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant services. In *NSDI*, 2007.

[23] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proc. SOSP*, 1991.

[24] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[25] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE TODSC*, 3(3):202–215, July 2006.

[26] S. Microsystems. NFS: Network file system protocol specification. `InternetRFC1094`, 1989.

[27] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *HPCA*, 2005.

[28] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *Proc. OSDI*, 2006.

[29] E. B. Nightingale, P. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, Oct. 2005.

[30] D. Openheimer, A. Ganapathi, and D. Patterson. Why do internet systems fail, and what can be done about it. In *Proc. USITS*, Seattle,WA, March 2003.

[31] M. Pease, R. Shostak, and L. Lamport. Reaching

agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.

[32] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. G. A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proc. SOSP*, 2005.

[33] R. Rodrigues, M. Castro, and B. Liskov. BASE : Using abstraction to improve fault tolerance. In *Proc. SOSP*, October 2001.

[34] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. OSDI*, December 1999.

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

[36] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. DSN*, 2002.

[37] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. Conflict-free quorum-based bft protocols. Technical Report 2007-1, Max Planck Institute for Software Systems, August 2007.

[38] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. ICDCS*, 2000.

[39] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proc. OSDI*, 2006.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proc. OSDI*, 2004.

[41] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proc. SOSP*, October 2003.

APPENDIX

The Zyzzyva view change sub-protocol is similar to traditional view change sub-protocols with two key exceptions. First, while replicas in traditional view change protocols commit to the view change as soon as they suspect the primary to be faulty, replicas in Zyzzyva only commit to a view change when they know that all other correct replicas will join them in electing a new primary. Second, Zyzzyva weakens the condition under which a request appears in the new view's history. The protocol 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 $\langle$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 faithfully participate 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 $\langle$I-HATE-THE-PRIMARY, $v\rangle_{\sigma_j}$ from $f+1$ distinct replicas j, and the message $\langle$VIEW-CHANGE$, v+1, CC, O, i\rangle_{\sigma_i}$ to all replicas. CC is either the most recent commit certificate for a request since the last view change, $f+1$ VIEW-CONFIRM messages if no commit certificate is available, or a NEW-VIEW message if neither of the previous options are available. O is i's ordered request history since the commit certificate indicated by CC. At this point, a replica stops accepting messages relevant to the current view and does not respond to the client until a new view has started.

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

Primary. Upon receipt of $2f+1$ VIEW-CHANGE messages, the new primary p constructs the message $\langle$NEW-VIEW$, v+1, P\rangle_{\sigma_p}$ where P is a collection of $2f+1$ VIEW-CHANGE messages defining the initial state for view $v+1$.

Replica. The replica starts a timer. If the 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 $v+2$.

<u>Additional Pedantic Details:</u> If a replica commits to change to view $v+2$ before receiving a new view message for view $v+1$, then the replica uses the set of ordered requests from view v to form its view change message. The length of the timer in the new view grows exponentially with the number view changes that fail in succession.

VC4. Replica receives a valid new view message and sends a view confirmation message to all other replicas.

Replicas determine the state of the new view based on the collection of $2f+1$ VIEW-CHANGE messages included in the NEW-VIEW message. The most recent request with a corresponding commit certificate (or old new view message) is accepted as the last request in the base history. The most recent request that is ordered subsequent to the commit certificate by at least $f+1$ VIEW-CHANGE messages is accepted. Replica i forms the message $\langle$VIEW-CONFIRM, $v+1$, n, h, $i\rangle_{\sigma_i}$ based on the NEW-VIEW message and sends the VIEW-CONFIRM message to all other replicas.

<u>Additional Pedantic Details:</u> When evaluating the NEW-VIEW message, a commit certificate from the most recent view takes priority over anything else, followed by $f+1$ VIEW-CONFIRM messages, and finally a NEW-VIEW message with the highest view number.

VC5. Replica receives $2f+1$ matching VIEW-CONFIRM messages and begins accepting requests in the new view.

Upon receipt of $2f+1$ matching VIEW-CONFIRM messages, replica i begins the new view v.

<u>Additional Pedantic Details:</u> The exchange of view confirm messages is not strictly necessary for safety and can be optimized out of the protocol, but including them simplifies our safety proof by ensuring that if a correct replica begins accepting messages in new view v, then no other correct replica will accept messages in view v with a different base history. This step allows replicas to consider a confirmed view change to be functionally equivalent to a commit certificate for all requests in the base history of the new view.