Zeno: Eventually Consistent Byzantine-Fault Tolerance

Atul Singh[1,2], Pedro Fonseca[1], Petr Kuznetsov[3], Rodrigo Rodrigues[1], Petros Maniatis[4]
[1]*MPI-SWS,* [2]*Rice University,*
[3]*TU Berlin/Deutsche Telekom Laboratories,* [4]*Intel Research Berkeley*
Technical Report MPI-SWS-2009-001

Abstract

Many distributed services are hosted at large, shared, geographically diverse data centers, and they use replication to achieve high availability despite the unreachability of an entire data center. Recent events show that non-crash faults occur in these services and may lead to long outages. While Byzantine-Fault Tolerance (BFT) could be used to withstand these faults, current BFT protocols can become unavailable if a small fraction of their replicas are unreachable. This is because existing BFT protocols favor strong safety guarantees (consistency) over liveness (availability).

This paper presents a novel BFT state machine replication protocol called Zeno that trades consistency for higher availability. In particular, Zeno replaces strong consistency (*linearizability*) with a weaker guarantee (*eventual consistency*): clients can temporarily miss each other's updates but when the network is stable the states from the individual partitions are merged by having the replicas agree on a total order for all requests. We have built a prototype of Zeno and our evaluation using micro-benchmarks shows that Zeno provides better availability than traditional BFT protocols.

1 Introduction

Data centers are becoming a crucial computing platform for large-scale Internet services and applications in a variety of fields. These applications are often designed as a composition of multiple services. For instance, Amazon's S3 storage service and its e-commerce platform use Dynamo [16] as a storage substrate, or Google's indices are built using the MapReduce [15] parallel processing framework, which in turn can use GFS [19] for storage.

Ensuring correct and continuous operation of these services is critical, since downtime can lead to loss of revenue, bad press, and customer anger [5]. Thus, to achieve high availability, these services replicate data and computation, commonly at multiple sites, to be able to withstand events that make an entire data center unreachable [16] such as network partitions, maintenance events, and physical disasters.

When designing replication protocols, assumptions have to be made about the types of faults the protocol is designed to tolerate. The main choice lies between a *crash-fault* model, where it is assumed nodes fail cleanly by becoming completely inoperable, or a *Byzantine-fault* model, where no assumptions are made about faulty components, capturing scenarios such as bugs that cause incorrect behavior or even malicious attacks. A crash-fault model is typically assumed in most widely deployed services today, including those described above; the primary motivation for this design choice is that all machines of such commercial services run in the trusted environment of the service provider's data center [16].

Unfortunately, the crash-fault assumption is not always valid even in trusted environments, and the consequences can be disastrous. To give a few recent examples, Amazon's S3 storage service suffered a multi-hour outage, caused by corruption in the internal state of a server that spread throughout the entire system [2]; also an outage in Google's App Engine was triggered by a bug in datastore servers that caused some requests to return errors [20]; and a multi-day outage at the Netflix DVD mail-rental was caused by a faulty hardware component that triggered a database corruption event [31].

Byzantine-fault-tolerant (BFT) replication protocols are an attractive solution for dealing with such faults. Recent research advances in this area have shown that BFT protocols can perform well in terms of throughput and latency [24], they can use a small number of replicas equal to their crash-fault counterparts [10, 39], and they can be used to replicate off-the-shelf, non-deterministic, or even distinct implementations of common services [32, 38].

However, most proposals for BFT protocols have focused on strong semantics such as linearizability [23], where intuitively the replicated system appears to the clients as a single, correct, sequential server. The price to pay for such strong semantics is that each operation must contact a large subset (more than $\frac{2}{3}$, or in some cases $\frac{4}{5}$) of the replicas to conclude, which can cause the system to halt if more than a small fraction ($\frac{1}{3}$ or $\frac{1}{5}$, respectively) of the replicas are unreachable due to maintenance events, network partitions, or other non-Byzantine faults. This contrasts with the philosophy of systems deployed in corporate data centers [16, 22, 36], which favor availability and performance, possibly sacrificing the semantics of the system, so they can provide continuous service and meet tight SLAs [16].

In this paper we propose Zeno, a new BFT replication protocol designed to meet the needs of modern services

running in corporate data centers. In particular, Zeno favors service performance and availability, at the cost of providing weaker consistency guarantees than traditional BFT replication when network partitions and other infrequent events reduce the availability of individual servers.

Zeno offers eventual consistency semantics [18], which intuitively means that different clients can be unaware of the effects of each other's operations, e.g., during a network partition, but operations are never lost and will eventually appear in a linear history of the service—corresponding to that abstraction of a single, correct, sequential server—once enough connectivity is re-established.

In building Zeno we did not start from scratch, but instead adapted Zyzzyva [24], a state-of-the-art BFT replication protocol, to provide high availability. Zyzzyva employs speculation to conclude operations fast and cheaply, yielding high service throughput during favorable system conditions—while connectivity and replicas are available—so it is a good candidate to adapt for our purposes. Adaptation was challenging for several reasons, such as dealing with the conflict between the client's need for a fast and meaningful response and the requirement that each request is brought to completion, or adapting the *view change* protocols to also enable progress when only a small fraction of the replicas are reachable and to merge the state of individual partitions when enough connectivity is re-established.

The rest of the paper is organized as follows. Section 2 motivates the need for eventual consistency. Section 3 defines the properties guaranteed by our protocol. Section 4 describe how Zeno works and Section 5 sketches the proof of its correctness. Section 6 evaluates how our implementation of Zeno performs. Section 7 presents related work, Section 8 concludes, and efficient symmetric cryptography based Zeno is presented in Appendix A.

2 The Case for Eventual Consistency

Various levels and definitions of weak consistency have been proposed by different communities [17], so we need to justify why our particular choice is adequate. We argue that eventual consistency is both *necessary* for the guarantees we are targetting, and *sufficient* from the standpoint of many applications.

Consider a scenario where a network partition occurs, that causes half of the replicas from a given replica group to be on one side of the partition and the other half on the other side. This is plausible given that replicated systems often spread their replicas over multiple data centers for increased reliability [16], and that Internet partitions do occur in practice [6]. In this case, eventual consistency is *necessary* to offer high availability to clients on both sides of the partition, since it is impossible to have both sides of the partitions make progress and simultaneously achieve a consistency level that provided a total order on the operations ("seen" by all client requests) [7]. Intuitively, the closest approximation from that idealized consistency that could be offered is eventual consistency, where clients on each side of the partition agree on an ordering (that only orders their operations with respect to each other), and, when enough connectivity is re-established, the two divergent states can be merged, meaning that a total order between the operations on both sides can be established, and subsequent operations will reflect that order.

Additionally, we argue that eventual consistency is *sufficient* from the standpoint of the properties required by many services and applications that run in data centers. This has been clearly stated by the designers of many of these services [3, 14, 16, 22, 36]. Applications that use an eventually consistent service have to be able to work with responses that may not include some previously executed operations. To give an example of applications that use Dynamo, this means that customers may not get the most up-to-date sales ranks, or may even see some items they deleted reappear in their shoping carts, in which case the delete operation may have to be redone. However, those events are much preferrable to having a slow, or unavailable service.

Beyond data-center applications, many other examples of eventually consistent services has been deployed in common-use systems, for example, DNS. Saito and Shapiro [33] provide a more thourough survey of the theme.

3 Algorithm Properties

We now informally specify safety and liveness properties of a generic eventually consistent BFT service.

To specify safety in a formal way, we use the language of I/O automata [29, chapter 8]. Our definitions extend the correctness criteria of a *linearizable* Byzantine-fault tolerant service [8], and a definition of a crash fault-tolerant eventually consistent service [18].

Figures 1 and 2 describe an I/O automaton corresponding to a generic eventually consistent service. We will use as a running example a shopping cart service with operations such as *AddItem*, *RemoveItem*, *CheckOut*, etc. The service is characterized by a set of *states* $\mathcal{Q}$ (all possible sets of items in a shopping cart), an initial state $q_0 \in \mathcal{Q}$ (an empty cart), a set of *clients* $\mathcal{C}$, a set of *servers* Π, a set of operations $\mathcal{O}$ (*AddItem*, *RemoveItem*, etc.), a set of responses $\mathcal{O}'$ (the result of an item addition or deletion) and a transition function $g : \mathcal{C} \times \mathcal{O} \times \mathcal{Q} \rightarrow \mathcal{O}' \times \mathcal{Q}$. This transition function models the sequential behavior of the state machine being replicated, for example, when a client invokes an *AddItem* operation for item x on an

Signature:

Inputs:	**Internals:**
REQUEST$(o,s)_c$	ENTER(o,s,t,c)
CLIENT-FAILURE$_c$	MERGE
SERVER-FAILURE$_i$	EXECUTE(o,s,t,c)
	FORK
Outputs:	FAULTY-REQUEST(o,s,t,c)
REPLY-WEAK$(r)_c$	COMMIT
REPLY-STRONG$(r)_c$	(Here, $o \in \mathcal{O}$, $c \in \mathcal{C}$, $t \in \mathbb{N}$, $i \in \Pi$, $r \in \mathcal{O}'$.)

State components:

invoked $\subseteq \mathcal{O} \times \{0,1\} \times \mathbb{N} \times \mathcal{C}$, initially empty
H, a set of *histories*, initially $\{\varepsilon\}$, ε being the empty history
out,*out-commit* $\subseteq \mathcal{O}' \times \mathbb{N} \times \mathcal{C}$, initially empty
$\forall c \in \mathcal{C}$, *last-req*$_c \in \mathbb{N}$, initially 0
$\forall c \in \mathcal{C}$, *faulty-client*$_c \in$ *Bool*, initially *false*
$\forall i \in \Pi$, *faulty-server*$_i \in$ *Bool*, initially *false*
failed $\equiv |\{i | \textit{faulty-server}_i = \textit{true}\}|$
maxhist $\equiv \lfloor (n - |\textit{failed}|)/(f + 1 - |\textit{failed}|) \rfloor$

Figure 1: Specification of an eventually consistent service: signature and state components.

empty shopping cart, the state changed to contain item *x*. We assume that at most $f < N = |\Pi|$ servers and any number of clients can be Byzantine faulty.

We model the global state of the eventually consistent service as a set, *invoked*, of operations, where each operation is equipped with a boolean flag, declaring whether the operation requires only a strong response (we call such a request *strong*), a timestamp (a non-negative integer), and a client identifier (*invoked* $\subseteq \mathcal{O} \times \{0,1\} \times \mathbb{N} \times \mathcal{C}$), a (multi)set, H, of *histories*, i.e., totally ordered subsets of *invoked*. The histories describe all possible concurrent views of the service state the non-faulty servers may have. This captures the intuitive notion that some operations may have executed without being aware of each other, e.g., on different sides of a network partition, and are therefore only ordered with respect to a subset of the requests that were executed. The service maintains an invariant that the total number of divergent histories never exceeds $\textit{maxhist} = \lfloor \frac{n-|\textit{failed}|}{f+1-|\textit{failed}|} \rfloor$, where $|\textit{failed}|$ is the current number of failed servers. Intuitively, with $|\textit{failed}|$ servers, there can be no more than *maxhist* groups of $f+1$ servers, disjoint on the set of correct servers. Each of these groups may therefore maintain an independent history of clients' requests.

A sequence of operations that are already committed are modelled as a map *committed*, defined on H, that maps each history $h \in H$ to a prefix of h. For each two histories h and h', *committed*(h) and *committed*(h') are related by containment, hence the order of every operation in *committed* does not change with time. For example, after a *CheckOut* operation on a shopping cart becomes committed, its position in the committed history becomes locked, and subsequent *CheckOut* operations that commit see the effects of the previous *CheckOut*.

Transitions:

CLIENT-FAILURE$_c$
Eff: *faulty-client*$_c$:= *true*

SERVER-FAILURE$_i$
Pre: $|\textit{failed}| < f$
Eff: *faulty-server*$_i$:= *true*

REQUEST$(o,s)_c$
Eff: *last-req*$_c$:= *last-req*$_c + 1$
invoked := *invoked* $\cup \{(o,s,\textit{last-req}_c,c)\}$

ENTER(o,s,t,c)
Pre: $(o,s,t,c) \in \textit{invoked} \wedge \exists h \in H : (o,s,t,c) \notin h$
Eff: $h :=$ select $h \in H : (o,s,t,c) \notin h$
add (o,s,t,c) to the end of h

MERGE
Pre: $|H| \geq 2$
Eff: select $\{h,h'\} \subseteq H$
$h'' :=$ merge h and h'
committed$(h'') := \max(\textit{committed}(h), \textit{committed}(h'))$
$H := H - \{h,h'\} + \{h''\}$

FORK
Pre: $|H| < \textit{maxhist}$
Eff: select $h \in H$
$H := H + \{h\}$

COMMIT
Eff: select $h \in H$ with the longest committed prefix
committed$(h) := h$

EXECUTE(o,s,t,c)
Pre: $\exists h \in H$: $(o,s,t,c) \in h$
Eff: select $h \in H$: $(o,s,t,c) \in h$
$r :=$ response of (o,s,t,c) in h
if $(o,s,t,c) \in \textit{committed}(h)$ then *out-commit* := *out-commit* $\cup \{(r,t,c)\}$
else if *not* s then *out* := *out* $\cup \{(r,t,c)\}$

REPLY-WEAK$(r)_c$
Pre: *faulty-client*$_c \vee$
$\exists t : (r,t,c) \in \textit{out}$
Eff: *out* := *out* $- \{(r,t,c)\}$

REPLY-STRONG$(r)_c$
Pre: *faulty-client*$_c \vee$
$\exists t : (r,t,c) \in \textit{out-commit}$
Eff: *out-commit* := *out-commit* $- \{(r,t,c)\}$

FAULTY-REQUEST$(o,s,t,c)_c$
Pre: *faulty-client*$_c$ = *true*
Eff: *in* := *in* $\cup \{(o,s,t,c)\}$

Figure 2: Specification of an eventually consistent service: transitions.

Faults of clients and replicas are modeled as input actions CLIENT-FAILURE$_c$ and SERVER-FAILURE$_i$, respectively.

To describe the transitions, we follow the flow of a client request. A request *o* generated by a correct client *c* is modelled as an input action REQUEST$(o)_c$ that computes the timestamp *t* of the current request of *c* and adds an element (o,s,t,c) to the set of invocations *invoked*. Internal action ENTER(o,s,t,c) adds the request (o,s,t,c) to the end of one of the histories in H, if the request was not already there.

Action MERGE creates a new history from two histories in H, which adopts the longest committed prefix of the two histories. Action FORK split a given history into two, under the condition that the total number of histories in H does not exceed *maxhist*. Intuitively, these two actions describe the evolution of views that correct servers may have in case of creation and merging of partitions.

At any time, one of the histories with the longest committed prefix can commit all its requests (action COMMIT).

A request (o,s,t,c) can be executed based on its position in a history h and the corresponding response is put in one of the output buffers *out* and *out-commit* (action EXECUTE).[1] If (o,s,t,c) is in *committed*(h), then a strong reply is put in *out-commit*. Otherwise, if the corresponding request was not declared strong, a weak reply is put in *out*. In our running example, an *AddItem* request for item x on an empty cart that is concurrent with another *AddItem* request for item y may receive a response in which only item x appears in the cart (for the case where *AddItem*(x) is eventually ordered before *AddItem*(y), or a response in which both x and y are in the cart (if *AddItem*(y) is ordered first).

Output actions REPLY-WEAK$(r)_c$ and REPLY-STRONG$(r)_c$ are enabled when, for some t, $(r,t,c) \in$ *out* or $(r,t,c) \in$ *out-commit*, respectively. When REPLY-WEAK$(r)_c$ is issued, we say that client c receives a *weak response*, and the corresponding request (o,s,t,c) is *weakly complete*. When REPLY-STRONG$(r)_c$ is issued, we say that client c receives a *strong response*, and (o,s,t,c) is *strongly complete* or *committed*.

The strong responses correspond to committed operations that are totally ordered, unlike the weak responses whose position in the total order is still undefined. In our example, adding an element might only wait for a weak response, but checking out could wait for a strong response, to ensure other checkout operations see that certain items were already checked out.

A system ensures *eventual consistency* if every trace it produces—that is, every sequence of input and output actions it exhibits as it evolves over time—belongs to the set of traces of the automaton in Figures 1 and 2.

3.1 Liveness

On the liveness side, our service guarantees that a request issued by a correct client is processed and a response is returned to the client, provided that the client can communicate with *enough* replicas in a timely manner.

More precisely, we assume a default round-trip delay Δ and we say that a set of servers $\Pi' \subseteq \Pi$, is *eventually synchronous* if there is a time after which every two-way message exchange within Π' takes at most Δ time units. We also assume that every two correct servers or clients can eventually reliably communicate. Now our progress requirements can be put as follows:

(L1) If there exists an eventually synchronous set of $f+1$ correct servers Π', then every weak request issued by a correct client is eventually weakly complete.

(L2) If there exists an eventually synchronous set of $2f+1$ correct servers Π', then every weakly complete request or a strong request issued by a correct client is eventually committed.

In particular, **(L1)** and **(L2)** imply that if there is a an eventually synchronous set of $2f+1$ correct replicas, then *each* (weak or strong) request issued by a correct client will eventually be committed.

As we will explain later, ensuring **(L1)** in the presence of partitions may require unbounded storage. We will present a protocol addition that bounds the storage requirements at the expense of relaxing **(L1)**.

4 Zeno Protocol

4.1 System model

Zeno is a BFT state machine replication protocol. It requires $N = (3f+1)$ replicas to tolerate f Byzantine faults, i.e., we make no assumption about the behavior of faulty replicas. Zeno also tolerates an arbitrary number of Byzantine clients. We assume no node can break cryptographic techniques like collision-resistant digests, encryption, and signing. The protocol we present in this paper uses public key digital signatures to authenticate communication. We present a modified version of the protocol that uses more efficient symmetric cryptography based on message authentication codes (MACs) in the Appendix A.

The protocol uses two kinds of quorums: *strong quorums* consisting of any group of $2f+1$ distinct replicas, and *weak quorums* of $f+1$ distinct replicas.

The system easily generalizes to any $N \geq 3f+1$, in which case the size of *strong quorums* becomes $\lceil \frac{N+f+1}{2} \rceil$, and weak quorums remain the same, independent of N. Note that one can apply our techniques in very large replica groups (where $N \gg 3f+1$) and still make progress as long as $f+1$ replicas are available, whereas traditional (strongly consistent) BFT systems can be blocked unless at least $\lceil \frac{N+f+1}{2} \rceil$ replicas, growing with N, are available.

4.2 Overview

Like most traditional BFT state machine replication protocols, Zeno has three components: *sequence number assignment* (Section 4.4) to determine the total order of operations, *view changes* (Section 4.5) to deal with leader replica election, and *checkpointing* (Section 4.8) to deal with garbage collection of protocol and application state.

The execution goes through a sequence of configurations called *views*. In each view, a designated leader replica (the *primary*) is responsible for assigning monotonically increasing sequence numbers to clients' operations. A replica j is the primary for the view numbered v iff $j = v \mod N$.

[1] We consider here deterministic services and, thus, a total order on a set of operations unambiguously determines the state of the service and the responses to all operations.

Name	Meaning
v	current view number
n	highest sequence number executed
h	history, a hash-chain digest of the requests
o	operation to be performed
t	timestamp assigned by the client to each request
s	flag indicating if this is a strong operation
r	result of the operation
$D(.)$	cryptographic digest function
CC	highest commit certificate
ND	non-deterministic argument to an operation
OR	Order Request message

Table 1: Notations used in message fields.

At a high level, normal case execution of a request proceeds as follows. A client first sends its request to all replicas. A designated primary replica assigns a sequence number to the client request and broadcasts this proposal to the remaining replicas. Then all replicas execute the request and return a reply to the client.

Once the client gathers sufficiently many *matching* replies—replies that agree on the operation result, the sequence number, the view, and the replica history—it returns this result to the application. For weak requests, it suffices that a single correct replica returned the result, since that replica will not only provide a correct weak reply by properly executing the request, but it will also eventually commit that request to the linear history of the service. Therefore, the client need only collect matching replies from a *weak quorum* of replicas. For strong requests, the client must wait for matching replies from a *strong quorum*, that is, a group of at least $2f+1$ distinct replicas. This implies that Zeno can complete many weak operations in parallel across different partitions when only weak quorums are available, whereas it can complete strong operations only when there are strong quorums available.

Whenever operations do not make progress, or if replicas agree that the primary is faulty, a view change protocol tries to elect a new primary. Unlike in previous BFT protocols, view changes in Zeno can proceed with the concordancy of only a weak quorum. This can allow multiple primaries to coexist in the system (e.g., during a network partition) which is necessary to make progress with eventual consistency. However, as soon as these multiple views (with possibly divergent sets of operations) detect each other (Section 4.6), they reconcile their operations via a merge procedure (Section 4.7), restoring consistency among replicas.

In what follows, messages with a subscript of the form σ_c denote a public-key signature by principal c. In all protocol actions, malformed or improperly signed messages are dropped without further processing. We interchangeably use terms "non-faulty" and "correct" to mean system components (e.g., replicas and clients) that follow our protocol faithfully. Table 1 collects our notation.

We start by explaining the protocol state at the replicas. Then we present details about the three protocol components. We used Zyzzyva [24] as a starting point for designing Zeno. Therefore, throughout the presentation, we will explain how Zeno differs from Zyzzyva.

4.3 Protocol State

Each replica i maintains the highest sequence number n it has executed, the number v of the view it is currently participating in, and an ordered history of requests it has executed along with the ordering received from the primary. Replicas maintain a hash-chain digest h_n of the n operations in their history in the following way: $h_{n+1} = D(h_n, D(\text{REQ}_{n+1}))$, where D is a cryptographic digest function and REQ_{n+1} is the request assigned sequence number $n+1$.

A prefix of the ordered history upto sequence number ℓ is called *committed* when a replica gathers a *commit certificate* (denoted *CC* and described in detail in Section 4.4) for ℓ; each replica only remembers the highest CC it witnessed.

To prevent the history of requests from growing without bounds, replicas assemble checkpoints after every *CHKP_INTERVAL* sequence numbers. For every checkpoint sequence number ℓ, a replica first obtains the *CC* for ℓ and executes all operations upto and including ℓ. At this point, a replica takes a snapshot of the application state and stores it (Section 4.8).

Replicas remember the set of operations received from each client c in their *request[c]* buffer and only the last reply sent to each client in their *reply[c]* buffer. The *request* buffer is flushed when a checkpoint is taken.

4.4 Sequence Number Assignment

To describe how sequence number assignment works, we follow the flow of a request.

Client sends request. A correct client c sends a request $\langle \text{REQUEST}, o, t, c, s \rangle_{\sigma_c}$ to all replicas, where o is the operation, t is a sequence number incremented on every request, and s is the strong operation flag.

Primary assigns sequence number and broadcasts order request (OR) message. If the last operation executed for this client has timestamp $t' = t-1$, then primary i assigns the next available sequence number $n+1$ to this request, increments n, and then broadcasts a $\langle \text{OR}, v, n, h_n, D(\text{REQ}), i, s, ND \rangle_{\sigma_i}$ message to backup replicas. *ND* is a set of non-deterministic application variables, such as a seed for a pseudorandom number generator, used by the application to generate non-determinism.

Replicas receive OR. When a replica j receives an OR message and the corresponding client request, it first checks if both are authentic, and then checks if it is in view v. If valid, it calculates $h'_{n+1} = D(h_n, D(\text{REQ}))$ and checks if h'_{n+1} is equal to the history digest in the OR message. Next, it increments its highest sequence number n, and executes the operation o from REQ on the application state and obtains a reply r. A replica sends the reply $\langle\langle \text{SPECREPLY}, v, n, h_n, D(r), c, t\rangle_{\sigma_j}, j, r, \text{OR}\rangle$ immediately to the client if s is *false* (i.e., this is a weak request). If s is *true*, then the request must be committed before replying, so a replica first multicasts a $\langle \text{COMMIT}, \text{OR}, j\rangle_{\sigma_j}$ to all others. When a replica receives at least $2f+1$ such COMMIT messages (including its own) matching in n, v, h_n, $D(\text{REQ})$, it forms a commit certificate *CC* consisting of the set of COMMIT messages and the corresponding OR, stores the *CC*, and sends the reply to the client in a message $\langle\langle \text{REPLY}, v, n, h_n, D(r), c, t\rangle_{\sigma_j}, j, r, \text{OR}\rangle$. The primary follows the same logic to execute the request, potentially committing it, and sending the reply to the client. Note that the commit protocol used for strong requests will also add all the preceding weak requests to the set of committed operations.

Client receives responses. For weak requests, if a client receives a weak quorum of SPECREPLY messages matching in their v, n, h, r, and OR, it considers the request weakly complete and returns a weak result to the application. For strong requests, a client requires matching REPLY messages from a strong quorum to consider the operation complete.

Fill Hole Protocol. Replicas only execute requests—both weak and strong—in sequence number order. However, due to message loss or other network disruptions, a replica i may receive an OR or a COMMIT message with a higher-than-expected sequence number (that is, $\text{OR}.n > n+1$); the replica discards such messages, asking the primary to "fill it in" on what it has missed (the OR messages with sequence numbers between $n+1$ and $\text{OR}.n$) by sending the primary a $\langle \text{FILLHOLE}, v, n, \text{OR}.n, i\rangle$ message. Upon receipt, the primary resends all of the requested OR messages back to i, to bring it up-to-date.

Comparison to Zyzzyva. There are four important differences between Zeno and Zyzzyva in the normal execution of the protocol.

First, Zeno clients only need matching replies from a weak quorum, whereas Zyzzyva requires at least a strong quorum; this leads to significant increase in availability, when for example only between $f+1$ and $2f$ replicas are available. It also allows for slightly lower overhead at the client due to reduced message processing requirements, and to a lower latency for request execution when internode latencies are heterogeneous.

Second, Zeno requires clients to use sequential timestamps instead of monotonically increasing but not necessarily sequential timestamps (which are the norm in comparable systems). This is required for garbage collection (Section 4.8). This raises the issue of how to deal with clients that reboot or otherwise lose the information about the latest sequence number. In our current implementation we are not storing this sequence number persistently before sending the request. We chose this because the guarantees we obtain are still quite strong: the requests that were already committed will remain in the system, this does not interfere with requests from other clients, and all that might happen is the client losing some of its initial requests after rebooting or oldest uncommitted requests. As future work, we will devise protocols for improving these guarantees further, or for storing sequence numbers efficiently using SSDs or NVRAM.

Third, whereas Zyzzyva offers a single-phase performance optimization, in which a request commits in only three message steps under some conditions (when all $3f+1$ replicas operate roughly synchronously and are all available and non-faulty), Zeno disables that optimization. The rationale behind this removal is based on the view change protocol (Section 4.5) so we defer the discussion until then. A positive side-effect of this removal is that, unlike with Zyzzyva, Zeno does not entrust potentially faulty clients with any protocol step other than sending requests and collecting responses.

Finally, clients in Zeno send the request to all replicas whereas clients in Zyzzyva send the request only to the primary replica. This change is required only in the MAC version of the protocol but we present it here to keep the protocol description consistent. At a high level, this change is required to ensure that a faulty primary cannot prevent a correct request that has weakly completed from committing—the faulty primary may manipulate a few of the MACs in an authenticator present in the request before forwarding it to others, and during commit phase, not enough correct replicas correctly verify the authenticator and drop the request. Interestingly, we find that the implementations of both PBFT and Zyzzyva protocols also require the clients to send the request directly to all replicas.

Our protocol description omits some of the pedantic details such as handling faulty clients or request retransmissions; these cases are handled similarly to Zyzzyva and do not affect the overheads or benefits of Zeno when compared to Zyzzyva.

4.5 View Changes

We now turn to the election of a new primary when the current primary is unavailable or faulty. The key point behind our view change protocol is that it must be able to proceed when only a weak quorum of replicas is available unlike view change algorithms in strongly consistent BFT systems which require availability of a strong quorum to make progress. The reason for this is the following: strongly consistent BFT systems rely on the *quorum intersection property* to ensure that if a strong quorum Q decides to change view and another strong quorum Q' decides to commit a request, there is at least one non-faulty replica in both quorums ensuring that view changes do not "lose" requests committed previously. This implies that the sizes of strong quorums are at least $2f+1$, so that the intersection of any two contains at least $f+1$ replicas, including—since no more than f of those can be faulty—at least one non-faulty replica. In contrast, Zeno does not require view change quorums to intersect; a weak request missing from a view change will be eventually committed when the correct replica executing it manages to reach a strong quorum of correct replicas, whereas strong requests missing from a view change will cause a subsequent provable divergence and application-state merge.

View Change Protocol. A client c retransmits the request to all replicas if it times out before completing its request. A replica i receiving a client retransmission first checks if the request is already executed; if so, it simply resends the SPECREPLY/REPLY to the client from its *reply[c]* buffer. Otherwise, the replica forwards the request to the primary and starts a IHateThePrimary timer.

In the latter case, if the replica does not receive an OR message before it times out, it broadcasts $\langle \text{IHATETHEPRIMARY}, v \rangle_{\sigma_i}$ to all replicas, but continues to participate in the current view. If a replica receives such accusations from a weak quorum, it stops participating in the current view v and sends a $\langle \text{VIEWCHANGE}, v+1, CC, \mathcal{O} \rangle_{\sigma_i}$ to other replicas, where CC is the highest commit certificate, and $\mathcal{O}$ is i's ordered request history since that commit certificate, i.e., all OR messages for requests with sequence numbers higher than the one in CC. It then starts the view change timer.

The primary replica j for view $v+1$ starts a timer with a shorter timeout value called the aggregation timer and waits until it collects a set of VIEWCHANGE messages for view $v+1$ from a *strong* quorum, or until its aggregation timer expires. If the aggregation timer expires and the primary replica has collected $f+1$ or more such messages, it sends a $\langle \text{NEWVIEW}, v+1, \mathcal{P} \rangle_{\sigma_j}$ to other replicas, where $\mathcal{P}$ is the set of VIEWCHANGE messages it gathered (we call this a *weak view change*, as opposed to one where a strong quorum of replicas participate which is called a *strong view change*). If a replica does not receive the NEWVIEW message before the view change timer expires, it starts a view change into the next view number.

Note that waiting for messages from a strong quorum is not needed to meet our eventual consistency specification, but helps to avoid a situation where some operations are not immediately incorporated into the new view, which would later create a divergence that would need to be resolved using our merge procedure. Thus it improves the availability of our protocol.

Each replica locally calculates the initial state for the new view by executing the requests contained in $\mathcal{P}$, thereby updating both n and the history chain digest h_n. The order in which these requests are executed and how the initial state for the new view is calculated is related to how we merge divergent states from different replicas, so we defer this explanation to Section 4.7. Each replica then sends a $\langle \text{VIEWCONFIRM}, v+1, n, h_n, i \rangle_{\sigma_i}$ to all others, and once it receives such VIEWCONFIRM messages matching in $v+1$, n, and h from a weak or a strong quorum (for weak or strong view changes, respectively) the replica becomes active in view $v+1$ and stops processing messages for any prior views.

The view change protocol allows a set of $f+1$ correct but slow replicas to initiate a global view change even if there is a set of $f+1$ synchronized correct replicas, which may affect our liveness guarantees (in particular, the ability to eventually execute weak requests when there is a synchronous set of $f+1$ correct servers). We avoid this by prioritizing client requests over view change requests as follows. Every replica maintains a set of client requests that it received but have not been processed (put in an ordered request) by the primary. Whenever a replica i receives a message from j related to the view change protocol (IHATETHEPRIMARY, VIEWCHANGE, NEWVIEW, or VIEWCONFIRM) for a higher view, i first forwards the outstanding requests to the current primary and waits until the corresponding ORs are received or a timer expires. For each pending request, if a valid OR is received, then the replica sends the corresponding response back to the client. Then i processes the original view change related messages from j according to the protocol described above. This guarantees that the system makes progress even in the presence of continuous view changes caused by the slow replicas in such pathological situations.

Comparison to Zyzzyva. View changes in Zeno differ from Zyzzyva in the size of the quorum required for a view change to succeed: we require $f+1$ view change messages before a new view can be announced, whereas previous protocols required $2f+1$ messages. Moreover,

the way a new view message is processed is also different in Zeno. Specifically, the start state in a new view must incorporate not only the highest *CC* in the VIEWCHANGE messages, but also all ORDERREQ that appear in any VIEWCHANGE message from the previous view. This guarantees that a request is incorporated within the state of a new view even if only a single replica reports it; in contrast, Zyzzyva and other similar protocols require support from a weak quorum for every request moved forward through a view change. This is required in Zeno since it is possible that only one replica supports an operation that was executed in a weak view and no other non-faulty replica has seen that operation, and because bringing such operations to a higher view is needed to ensure that weak requests are eventually committed.

The following sections describe additions to the view change protocols to incorporate functionality for detecting and merging concurrent histories, which are also exclusive to Zeno.

4.6 Detecting Concurrent Histories

Concurrent histories (i.e., divergence in the service state) can be formed for several reasons. This can occur when the view change logic leads to the presence of two replicas that simultaneously believe they are the primary, and there are a sufficient number of other replicas that also share that belief and complete weak operations proposed by each primary. This could be the case during a network partition that splits the set of replicas into two subsets, each of them containing at least $f+1$ replicas.

Another possible reason for concurrent histories is that the base history decided during a view change may not have the latest committed operations from prior views. This is because a view change quorum (a weak quorum) may not share a non-faulty replica with prior commitment quorums (strong quorums) and remaining replicas; as a result, some committed operations may not appear in VIEWCHANGE messages and, therefore, may be missing from the new starting state in the NEWVIEW message.

Finally, a misbehaving primary can also cause divergence by proposing the same sequence numbers to different operations, and forwarding the different choices to disjoint sets of replicas.

Basic Idea. Two request history orderings $h_1^i, h_2^i, \ldots$ and $h_1^j, h_2^j, \ldots$, present at replicas i and j respectively, are called *concurrent* if there exists a sequence number n such that $h_n^i \neq h_n^j$; because of the collision resistance of the hash chaining mechanism used to produce history digests, this means that the sequence of requests represented by the two digests differ as well. A replica compares history digests whenever it receives protocol messages such as OR, COMMIT, or CHECKPOINT (described in Section 4.8) that purport to share the same history as its own.

For clarity, we first describe how we detect divergence within a view and then discuss detection across views. We also defer details pertaining to garbage collection of replica state until Section 4.8.

4.6.1 Divergence between replicas in same view

Suppose replica i is in view v_i, has executed up to sequence number n_i, and receives a properly authenticated message $\langle \text{OR}, v_i, n_j, h_{n_j}, D(\text{REQ}), p, s, ND \rangle_{\sigma_p}$ or $\langle \text{COMMIT}, \langle \text{OR}, v_i, n_j, h_{n_j}, D(\text{REQ}), p, s, ND \rangle_{\sigma_p}, j \rangle_{\sigma_j}$ from replica j.

If $n_i < n_j$, i.e., j has executed a request with sequence number n_j, then the fill-hole mechanism is started, and i receives from j a message $\langle \text{OR}, v', n_i, h_{n_i}, D(\text{REQ}'), k, s, ND \rangle_{\sigma_k}$, where $v' \leq v_i$ and $k = primary(v')$.

Otherwise, if $n_i \geq n_j$, both replicas have executed a request with sequence number n_j and therefore i must have the some $\langle \text{OR}, v', n_j, h_{n_j}, D(\text{REQ}'), k, s, ND \rangle_{\sigma_k}$ message in its log, where $v' \leq v_i$ and $k = primary(v')$.

If the two history digests match (the local h_{n_j} or h_{n_i}, depending on whether $n_i \geq n_j$, and the one received in the message), then the two histories are consistent and no concurrency is deduced.

If instead the two history digests differ, the histories must differ as well. If the two OR messages are authenticated by the same primary, together they constitute a *proof of misbehavior (POM)*; through an inductive argument it can be shown that the primary must have assigned different requests to the same sequence number n_j. Such a POM is sufficient to initiate a view change and a merge of histories (Section 4.7).

The case when the two OR messages are authenticated by different primaries indicates the existence of divergence, caused for instance by a network partition, and we discuss how to handle it next.

4.6.2 Divergence across views

Now assume that replica i receives a message from replica j indicating that $v_j > v_i$. This could happen due to a partition, during which different subsets changed views independently, or due to other network and replica asynchrony. Replica i requests the NEWVIEW message for v_j from j. (The case where $v_j < v_i$ is similar, with the exception that i pushes the NEWVIEW message to j instead.)

When node i receives and verifies the $\langle \text{NEWVIEW}, v_j, \mathcal{P} \rangle_{\sigma_p}$ message, where p is the issuing primary of view v_j, it compares its local history to the sequence of OR messages obtained after ordering the OR message present in the NEWVIEW message

(according to the procedure described in Section 4.7). Let n_l and n_h be the lowest and highest sequence numbers of those OR messages, respectively.

Case 1: [$n_i < n_l$] Replica i is missing future requests, so it sends j a FILLHOLE message requesting the OR messages between n_i and n_l. When these are received, it compares the OR message for n_i to detect if there was divergence. If so, the replica obtained a *proof of divergence (POD)*, consisting of the two OR messages, which it can use to initiate a new view change. If not, it executes the operations from n_i to n_l and ensures that its history after executing n_l is consistent with the *CC* present in the NEWVIEW message, and then handles the NEWVIEW message normally and enters v_j. If the histories do not match this also constitutes a POD.

Case 2: [$n_l \leq n_i \leq n_h$] Replica i must have the corresponding ORDERREQ for all requests with sequence numbers between n_l and n_i and can therefore check if its history diverges from that which was used to generate the new view. If it finds no divergence, it moves to v_j and calculates the start state based on the NEWVIEW message (Section 4.5). Otherwise, it generates a *POD* and initiates a merge.

Case 3: [$n_i > n_h$] Replica i has corresponding OR messages for all sequence numbers appearing in the NEWVIEW and can check for divergence. If no divergence is found, the replica has executed more requests in a lower view v_i than v_j. Therefore, it generates a *Proof of Absence (POA)*, consisting of all OR messages with sequence numbers in $[n_l, n_i]$ and the NEWVIEW message for the higher view, and initiates a merge. If divergence is found, i generates a *POD* and also initiates a merge.

Like traditional view change protocols, a replica i does not enter v_j if the NEWVIEW message for that view did not include all of i's committed requests. This is important for the safety properties providing guarantees for strong operations, since it excludes a situation where requests could be committed in v_j without seeing previously committed requests.

4.7 Merging Concurrent Histories

Once concurrent histories are detected, we need to merge them in a deterministic order. The solution we propose is to extend the view change protocol, since many of the functionalities required for merging are similar to those required to transfer a set of operations across views.

We extend the view change mechanism so that view changes can be triggered by either PODs, POMs or POAs. When a replica obtains a POM, a POD, or a POA after detecting divergence, it multicasts a message of the form $\langle \text{POM}_{\text{MSG}}, v, POM \rangle_{\sigma_i}$, $\langle \text{POD}_{\text{MSG}}, v, POD \rangle_{\sigma_i}$, or $\langle \text{POA}_{\text{MSG}}, v, POA \rangle_{\sigma_i}$ in addition to the VIEWCHANGE message for v. Note here that v in *POM* and *POD* is one higher than the highest view number present in the conflicting ORDERREQ messages, or one higher than the view number in the NEWVIEW component in the case of a *POA*.

Upon receiving an authentic and valid POM$_{\text{MSG}}$ or POD$_{\text{MSG}}$ or a POA$_{\text{MSG}}$, a replica broadcasts a VIEWCHANGE along with the triggering POM, POD, or POA message.

The view change mechanism will eventually lead to the election of a new primary that is supposed to multicast a NEWVIEW message. When a node receives such a message, it needs to compute the start state for the next view based on the information contained in that message. The new start state is calculated by first identifying the highest *CC* present among all VIEWCHANGE messages; this determines the new base history digest h_n for the start sequence number n of the new view.

But nodes also need to determine how to order the different OR messages that are present in the NEWVIEW message but not yet committed. Contained OR messages (potentially including concurrent requests) are ordered using a deterministic function of the requests that produces a total order for these requests. Having a fixed function allows all nodes receiving the NEWVIEW message to easily agree on the final order for the concurrent OR present in that message. Alternatively, we could let the primary replica propose an ordering, and disseminate it as an additional parameter of the NEWVIEW message.

Replicas receiving the NEWVIEW message then execute the requests in the OR messages according to that fixed order, updating their histories and history digests. If a replica has already executed some weak operations in an order that differs from the new ordering, it first rolls back the application state to the state of the last checkpoint (Section 4.8) and executes all operations after the checkpoint, starting with committed requests and then with the weak requests ordered by the NEWVIEW message. Finally, the replica broadcasts a VIEWCONFIRM message. As mentioned, when a replica collects matching VIEWCONFIRM messages on v, n, and h_n it becomes active in the new view.

Our merge procedure re-executes the concurrent operations sequentially, without running any additional or alternative application-specific conflict resolution procedure. This makes the merge algorithm slightly simpler, but requires the application upcall that executes client operations to contain enough information to identify and resolve concurrent operations. This is similar to the design choice made by Bayou [35] where special concurrency detection and merge procedure are part of each service operation, enabling servers to automatically detect and resolve conflicts.

Limiting the number of merge operations. A faulty replica can trigger multiple merges by producing a new POD for each conflicting request in the same view, or generating PODs for requests in old views where itself or a colluding replica was the primary. To avoid this potential performance problem, replicas remember the last POD, POM, or a POA every other replica initiated, and reject a POM/POD/POA from the same or a lower view coming from that replica. This ensures that a faulty replica can initiate a POD/POM/POA only once from each view it participated in. This, as we show in Section 5, helps establish our liveness properties.

Recap comparison to Zyzzyva. Zeno's view changes motivate our removal of the single-phase Zyzzyva optimization for the following reason: suppose a strong client request REQ was executed (and committed) at sequence number n at $3f+1$ replicas. Now suppose there was a weak view change, the new primary is faulty, and only $f+1$ replicas are available. A faulty replica among those has the option of reporting REQ in a different order in its VIEWCHANGE message, which enables the primary to order REQ arbitrarily in its NEWVIEW message; this is possible because only a single—potentially faulty—replica need report any request during a Zeno view change. This means that linearizability is violated for this strong, committed request REQ. Although it may be possible to design a more involved view change to preserve such orderings, we chose to keep things simple instead. As our results show, in many settings where eventual consistency is sufficient for weak operations, our availability under partitions tramps any benefits from increased throughput due to the Zyzzyva's optimized single-phase request commitment.

4.8 Garbage Collection

The protocol we have presented so far has two important shortcomings: the protocol state grows unboundedly, and weak requests are never committed unless they are followed by a strong request.

To address these issues, Zeno periodically takes checkpoints, garbage collecting its logs of requests and forcing weak requests to be committed.

When a replica receives an ORDERREQ message from the primary for sequence number M, it checks if M mod CHKP_INTERVAL $= 0$. If so, it broadcasts the COMMIT message corresponding to M to other replicas. Once a replica receives $2f+1$ COMMIT messages matching in v, M, and h_M, it creates the commit certificate for sequence number M. It then sends a $\langle \text{CHECKPOINT}, v, M, h_M, App \rangle_{\sigma_j}$ to all other replicas. The *App* is a snapshot of the application state after executing requests upto and including M. When it receives $f+1$ matching CHECKPOINT messages, it considers the checkpoint stable, stores this proof, and discards all ordered requests with sequence number lower than n along with their corresponding client requests.

Also, in case the checkpoint procedure is not run within the interval of T_{CHKP} time units, and a replica has some not yet committed ordered requests, the replica also initiates the commit step of the checkpoint procedure. This is done to make sure that pending ordered requests are committed when the service is rarely used by other clients and the sequence numbers grow very slowly.

Our checkpoint procedure described so far poses a challenge to the protocol for detecting concurrent histories. Once old requests have been garbage-collected, there is no way to verify, in the case of a slow replica (or a malicious replica pretending to be slow) that presents an old request, if that request has been committed at that sequence number or if there is divergence.

To address this, clients send sequential timestamps to uniquely identify each one of their own operations, and we added a list of per-client timestamps to the checkpoint messages, representing the maximum operation each client has executed up to the checkpoint. This is in contrast with previous BFT replication protocols, including Zyzzyva, where clients identified operations using timestamps obtained by reading their local clocks. Concretely, a replica sends $\langle \text{CHECKPOINT}, v, M, h_M, App, CSet \rangle_{\sigma_j}$, where *CSet* is a vector of $\langle c, t \rangle$ tuples, where t is the timestamp of the last committed operation from c.

This allows us to detect concurrent requests, even if some of the replicas have garbage-collected that request. Suppose a replica i receives an OR with sequence number n that corresponds to client c's request with timestamp t_1. Replica i first obtains the timestamp of the last executed operation of c in the highest checkpoint t_c=*CSet*[c]. If $t_1 \leq t_c$, then there is no divergence since the client request with timestamp t_1 has already been committed. But if $t_1 > t_c$, then we need to check if some other request was assigned n, providing a proof of divergence. If $n < M$, then the CHECKPOINT and the OR form a POD since some other request was assigned n. Else, we can perform regular conflict detection procedure to identify concurrency (see Section 4.6).

Note that our checkpoints become stable only when there are at least $2f+1$ replicas that are able to agree. In the presence of partitions or other unreachability situations where only weak quorums can talk to each other, it may not be possible to gather a checkpoint, which implies that Zeno must either allow the state concerning tentative operations to grow without bounds, or weaken its liveness guarantees. In our current protocol we chose the latter, and so replicas stop participating once they reach a maximum number of tentative operations they can execute, which could be determined based on their available storage resources (memory as well as the disk

space). Garbage collecting weak operations and the resulting impact on conflict detection is left as a future work.

5 Correctness

In this section, we sketch the proof that Zeno satisfies the safety properties specified in Section 3.

In Zeno, a (weak or strong) response is based on identical histories of at least $f+1$ replicas, and, thus, at least one of these histories belongs to a correct replica. Hence, in the case that our garbage collection scheme is not initiated, we can reformulate the safety requirements as follows: **(S1)** the local history maintained by a correct replica consists of a prefix of committed requests extended with a sequence of speculative requests, where no request appears twice, **(S2)** a request associated with a correct client c appears, in a history at a correct replica only if c has previously issued the request, and **(S3)** the committed prefixes of histories at every two correct replicas are related by containment, and **(S4)** at any time, the number of conflicting histories maintained at correct replica does not exceed $maxhist = \lfloor (N-f')/(f-f'+1) \rfloor$, where f' is the number of currently failed replicas and N is the total number of replicas required to tolerate a maximum of f faulty replicas. Here we say that two histories are conflicting if none of them is a prefix of the other.

Properties **(S1)** and **(S2)** are implied by the state maintenance mechanism of our protocol and the fact that only properly signed requests are put in a history by a correct replica. The special case when a prefix of a history is hidden behind a checkpoint is discussed later.

A committed prefix of a history maintained at a correct replica can only be modified by a commitment of a new request or a merge operation. The sub-protocol of Zeno responsible for committing requests are analogous to the two-phase conservative commitment in Zyzzyva [24], and, similarly, guarantees that all committed requests are totally ordered. When two histories are merged at a correct replica, the resulting history adopts the longest committed prefix of the two histories. Thus, inductively, the committed prefixes of all histories maintained at correct replicas are related by containment **(S3)**.

Now suppose that at a given time, the number of conflicting histories maintained at correct replica is more than *maxhist*. Our weak quorum mechanism guarantees that each history maintained at a correct process is supported by at least $f+1$ distinct processes (through sending SPECREPLY and REPLY messages). A correct process cannot concurrently acknowledge two conflicting histories. But when f' replicas are faulty, there can be at most $\lfloor (n-f')/(f-f'+1) \rfloor$ sets of $f+1$ replicas that are disjoint in the set of correct ones. Thus, at least one correct replica acknowledged two conflicting histories — a contradiction establishes **(S4)**.

Checkpointing. Note that our garbage collection scheme may affect property **(S1)**: the sequence of tentative operations maintained at a correct replica may potentially include a committed but already garbage-collected operation. This, however, cannot happen: each round of garbage collection produces a checkpoint that contains the latest committed service state and the logical timestamp of the latest committed operation of every client. Since no correct replica agrees to commit a request from a client unless its previous requests are already committed, the checkpoint implies the set of timestamps of all committed requests of each client. If a replica receives an ordered request of a client c corresponding to a sequence number preceding the checkpoint state, and the timestamp of this request is no later than the last committed request of c, then the replica simply ignores the request, concluding that the request is already committed. Hence, no request can appear in a local history twice.

5.1 Liveness

To show that Zeno complies with the liveness properties, we first establish that every weak request issued by a correct client is complete if a weak quorum of correct replicas is eventually synchronous, and then we show how the existence of a strong eventually synchronous quorum implies that each weakly complete request or a strong request issued by a correct client is committed. We assume that correct replicas and clients can eventually reliably communicate.

5.1.1 Weak partitions

Consider Π', an eventually synchronous set of $f+1$ correct replicas and let client c issue a weak request $(o,0,t,c)$. By contradiction, suppose that property **(L1)** does not hold, i.e., the client's request never completes. By the client's protocol, c keeps periodically rebroadcasting the request until it receives at least $f+1$ matching responses from distinct replicas. Recall that correct clients are well formed. Thus, c has earlier completed all requests with timestamps $t' < t$. Since every complete request involves at least one correct replica, we know that at least one correct replica stored the corresponding ordered requests in its local history. Periodic exchanges of SPECREPLY messages (e.g., during checkpoints) ensure that all such ordered requests will eventually reach every correct replica in Π'.

Now consider time t_0 after which Π' becomes synchronous, i.e., after t_0, round-trip delays between every two correct replicas in Π' do not exceed Δ.

View changes for non-responsive partitions. By the replica's protocol, each correct replica that receives the

request forwards it to the primary of its current view and sets the timer. If the timer expires before the corresponding ordered request is received from the primary (which can only happen after t_0 if the primary is faulty), the replica initiates a view change protocol. If no correct replica in Π' received the ordered request on time, then at least $f+1$ correct replicas in Π' will commit to a view change. Otherwise, if at least one correct replica in Π' received the ordered request, then every correct replica will receive the ordered request in the current view since correct replicas can obtain the OR corresponding to a request from other correct replicas. Thus, either every correct replica in Π' will send a speculative response to c (based on the history proposed by the primary), or every correct replica in Π' will switch to the next view. In the former case, since c never completes its request, $f+1$ speculative responses based on the ordered request from the same view and received by c do not match, which constitutes a POM (proof of misbehavior) against the primary and will be used as a basis for a view change.

Thus, as long as c does not complete its operation, the partition goes through consequent view changes. Now we need to show that the view changes do not indefinitely prevent replicas in Π' from making progress. We observe first that, since primaries are assigned to views in a round-robin fashion, the correct replicas in Π' will eventually reach a view whose primary is correct. But what about faulty or stale replicas with conflicting histories? Can they cause a view change even when the current primary is correct? In fact, the answer is yes, but we show below that they can only cause a bounded number of such view changes.

View changes for divergent histories. When a replica i finds out that another replica j is in a fresher view v, i adopts v as its view and tries to reconcile the histories. If, after adopting the most recent commit certificate n, i realizes that the same sequence number $n' > n$ is assigned to two different (not yet committed) requests at i and j, it can use this as an *evidence* for a view change.

However, such evidence can be used at most once per view: different requests assigned to the same sequence number by the same primary (or different primary) appear as a POM, POA, or POD. Furthermore, once two divergent histories from i and j are used as evidence for a view change in a given view v (these divergent histories are part of POM, POA, or or POD message), histories of i and j in previous views are ignored. Thus, eventually, faulty and stale replicas will run out of pieces of evidence and the correct replicas in Π' will reach a view with a correct primary in Π'. Since each correct replica maintains a set of received but not yet processed clients' requests that eventually include the request of c, and always tries to process the requests before taking part in a view change, the outstanding request of c will reach the primary in Π', the primary will generate an ordered request that will be sustained by at least $f+1$ replicas in a timely manner, and at least $f+1$ matching responses will reach the client — a contradiction. Hence, **(L1)** is ensured.

5.1.2 *Strong and weakly complete requests*

Finally, consider time after which a "strong" quorum Π' containing $2f+1$ correct replicas becomes synchronous. Assume, by contradiction, that some strong or weakly complete request (o,s,t,c) never gets committed. First, since a correct client keeps retransmitting a request until it is complete, our protocol ensures that an ordered request containing (o,s,t,c) is eventually adopted by some correct replica. Eventually, the ordered request will reach every correct replica in Π', e.g., during checkpointing or merge operations. As long as the set of ordered but not yet committed requests is not empty, correct replicas will periodically initiate the checkpointing sub-protocol in order to commit these requests. As we showed above, all correct replicas in the partition will eventually stabilize at the same view with a correct primary. Thus, eventually, $2f+1$ correct replicas will succeed in committing all outstanding ordered requests proposed by the primary and these requests will include (o,t,c)—a contradiction. Hence, **(L2)** is ensured.

6 Evaluation

We have implemented a prototype of Zeno as an extension to the publicly available Zyzzyva source code [26].

Our evaluation tries to answer the following questions: (1) Does Zeno incur more overhead than existing protocols in the normal case? (2) Does Zeno provide higher availability compared to existing protocols when there are more than f unreachable nodes? (3) What is the cost of merges?

Experimental setup. We set $f=1$, and the minimum number of replicas to tolerate it, $N=3f+1=4$. We vary the number of clients to increase load. Each physical machine has a dual-core 2.8 GHz AMD processor with 4GB of memory, running a 2.6.20 Linux kernel. Each replica as well as a client runs on a dedicated physical machine. We use Modelnet [37] to simulate a network topology consisting of two hubs connected via a bi-directional link unless otherwise mentioned. Each hub has two servers in all of our experiments but client location varies as per the experiment. Each link has one-way latency of 1 ms and a 100 Mbps bandwidth.

Transport protocols. Zyzzyva, like PBFT, uses multicast to reduce the cost of sending operations from clients to all replicas, so it uses UDP as a transport protocol and

implements a simple backoff and retry policy to handle message loss. This is not optimized for periods of congestion and high message loss, such as those we anticipate during merges when the replicas that were partitioned need to bring each other up-to-date. To address this, Zeno uses TCP as the transport layer during the merge procedure but continues to use Zyzzyva's UDP-based transport during normal operation and multicasting communication that is sent to all replicas.

Partition. We simulate network partitions by separating the two hubs from each other. We vary the duration of the partitions from 1 to 5 minutes, based on the observation by Chandra et al. [13] that a large fraction ($> 75\%$) of network disconnectivity events range from 30 to 500 seconds.

6.1 Implementation

Replacing PKI with MACs. Our Zeno prototype uses MACs instead of the slower digital signatures to implement message authentication for the common-case, but still uses signatures for view changes. Using MACs induces some small mechanistic design changes over the protocol description in Section 4; these changes are standard practice in similar protocols including Zyzzyva, and are presented in Appendix A.

Merge. Replicas detect divergence by following the algorithm specified in Section 4.7. We implemented an optimization to the merge protocol where replicas first move to the higher view and then propagate their local uncommitted requests to the primary of the higher view. The primary of the higher view orders these requests as if they are received from the client and hence merges these requests in the history.

6.2 Results

We generate a workload with a varying fraction of strong and weak operations. If each client issued both strong and weak operations, then most clients would block soon after network partitions started. Instead, we simulate two kind of clients: (i) weak clients only issue weak requests and (ii) strong clients always pose strong requests. This allows us to vary the ratio of weak operations (denoted by α) in the total workload with a limited number of clients in the system and long network partitions. We use a micro-benchmark that executes a no-op when the *execute* upcall for the client operation is invoked.

We have also built a simple application on top of Zeno, emulating a shopping cart service with operations to add, remove, and checkout items based on a *key-value* data store. We also implement a simple conflict detection and merge procedure.

Protocol	Batch=1	Batch=10
Zyzzyva (single phase)	62 Kops/s	88 Kops/s
Zeno (weak)	60 Kops/s	86 Kops/s
Zeno (strong)	40 Kops/s	82 Kops/s
Zyzzyva (commit opt)	40 Kops/s	82 Kops/s

Table 2: Peak throughput of Zeno and Zyzzyva.

6.2.1 Maximum throughput in the normal case

We compare the normal case performance of Zeno with Zyzzyva. In both systems we used the optimization of batching requests to reduce protocol overhead. In this experiment, the clients and servers are connected by a 1 Gbps switch with 0.1 ms round trip latency. We expect the peak throughput of Zeno with weak operations to approximately match the peak throughput of Zyzzyva since both can be completed in a single phase. However, the performance of Zeno with strong operations will be lower than the peak throughput of Zyzzyva since Zeno requires an extra phase to commit a strong operation.

Our results presented in Table 2 show that Zeno and Zyzzyva's throughput are similar, with Zyzzyva achieving slightly (3–6%) higher throughput than Zeno's throughput for weak operations. The results also show that, with batching, Zeno's throughput for strong operations is also close to Zyzzyva's peak throughput: Zyzzyva has 7% higher throughput when the single phase optimization is employed. However, when a single replica is faulty or slow, Zyzzyva cannot achieve the single phase throughput and Zeno's throughput for strong operations is identical to Zyzzyva's performance with a faulty replica.

6.2.2 Partition with no concurrency

For all the remaining experiments, we use Modelnet setup and disable multicast since Modelnet does not support it. We use a client population of 4 nodes, each sending a new request of minimal payload (2 Bytes) as soon as it has completed the previous request. This generates a steady load of approximately 500 requests/sec on the system. This is similar to an example SLA provided in Dynamo [16]. We use a batch size of 1 for both Zyzzyva and Zeno, since it is sufficient to handle the incoming request load.

In this experiment, all clients reside in the first LAN. We initiate a partition at 90 seconds which continues for a minute. Since there are no clients in the second LAN, there are no requests processed in it and hence there is no concurrency, which avoids the cost of merging. Replicas with id 0 (primary for view initial view 0) and 1 reside in the first LAN while replicas with ids 2 and 3 reside in the second LAN. We also present the results of Zyzzyva to compare the performance in both normal cases as well as under the given failure.

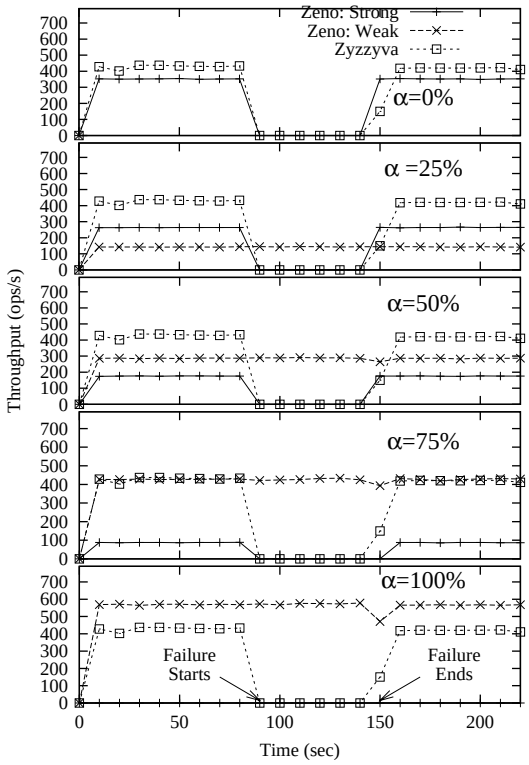

Figure 3: Two replicas are disconnected via a partition, that starts at time 90 and continues for 60 seconds. Parameter α represents the fraction of weak operations in the workload. Note that the throughput of weak and strong operations in Zeno is presented separately for clarity.

Varying α. We vary the mix of weak and strong operations in the workload, and present the results in Figure 3. First, strong operations block as soon as the failure starts which is expected since not enough replicas are reachable from the first LAN to complete the strong operation. However, as soon as the partition heals, we observe that strong operations start to be completed. Note also that Zyzzyva also blocks as soon as the failure starts and resumes as soon as it ends.

Second, weak operations continue to be processed and completed during the partition and this is because Zeno requires (for $f = 1$) only 2 non-faulty replicas to complete the operation. The fraction of total requests completed increases as α increases, essentially improving the availability of such operations despite network partitions.

Third, when replicas in the other LAN are reachable again, they need to obtain the missing requests from the first LAN. Since the number of weak operations performed in the first LAN increases as α increases, the time to update the lagging replicas in the other partition also goes up; this puts a temporary strain on the network, evidenced by the dip in the throughput of weak operations when the partition heals. However, this dip is brief compared to the duration of the partition. We explore the impact of the duration of partitions next.

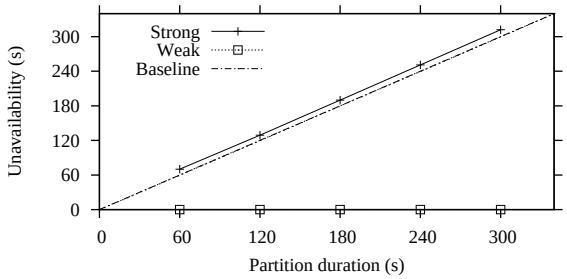

Figure 4: Varying partition durations with no concurrent operations. Baseline represents the minimal unavailability expected for strong operations, which is equal to the partition duration.

Varying partition duration. Using the same setup, we now vary partition durations between 1 and 5 minutes for $\alpha = 75\%$. For each partition duration, we measure the period of unavailability for both weak and strong operations. The unavailability is measured as the number of seconds for which the observed throughput, on either side of the partition, was less than 10% of the average throughput observed before the partition started. Also, the distance from the "Strong" line to the baseline ($x = y$) indicates how soon after healing the partition can strong operations be processed again.

Figure 4 presents the results. We observe that weak operations are always available in this experiment since all weak operations were completed in the first LAN and the replicas in the first LAN are up-to-date with each other to process the next weak operation. Strong operations are unavailable for the entire duration of the partition due to unavailability of the replicas in the second LAN and the additional unavailability is introduced by Zeno due to the operation transfer mechanism. However, the additional delay is within 4% of the partition duration (12 seconds for a 5 minute partition). Our current prototype is not yet optimized and we believe that the delay could be further reduced.

Varying request size. In this experiment, we simulate a partition for 60 seconds but increase the payload sizes from 2 Bytes to 1 KB, with an equally sized reply. The cumulative bandwidth of requests to be transferred from one LAN to the other is a function of the weak request offered load, the size of the requests, and the duration of the partition. With 60 seconds of partition and an offered load of 500 req/s, the cumulative request payload ranges from approximately 60 KB to 30 MB for 2 Bytes and 1 KB request size respectively. The results we obtained are very similar to those in Figure 3 so we do not repeat them. These show that the time to bring replicas in the second LAN up-to-date does not increase significantly

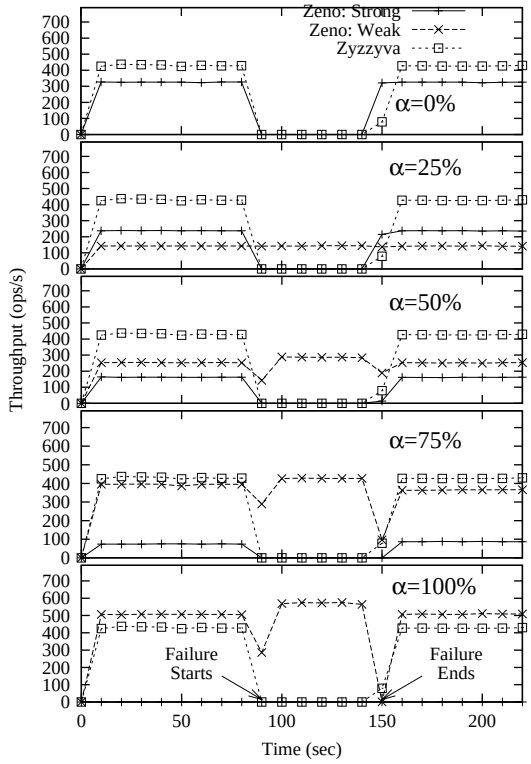

Figure 5: Network partition for 60 seconds starting at time 90 seconds. Note that the throughput of weak and strong operations in Zeno is presented separately for clarity.

with the increase in request size. Given that we have 100 Mbps links connecting replicas to each other, bandwidth is not a limiting resource for shipping operations at these offered loads.

6.2.3 Partition with concurrency

In this experiment, we keep half the clients on each side of a partition. This ensures that both partitions observe a steady load of weak operations that will cause Zeno to first perform a weak view change and later merge the concurrent weak operations completed in each partition. Hence, this microbenchmark additionally evaluates the cost of weak view changes and the merge procedure. As before, the primary for the initial view resides in the first LAN. We measure the overall throughput of weak and strong operations completed in both partitions. Again, we compare our results to Zyzzyva.

Varying α. Figure 5 presents the results for the throughput of different systems while varying the value of α. We observe three main points.

When $\alpha = 0$, Zeno does not give additional benefits since there are no weak operations to be completed. Also, as soon as the partition starts, strong operations are blocked and resume after the partition heals. As above, Zyzzyva provides greater throughput thanks to its single-phase execution of client requests, but it is as powerless to make progress during partitions as Zeno in the face of strong operations only.

When $\alpha = 25\%$, we have only one client sending *weak* operations in one LAN. Since there are no conflicts, this graph matches that of Figure 3.

When $\alpha \geq 50\%$, we have at least two weak clients, at least one in each LAN. When a partition starts, we observe that the throughput of weak operations first drops; this happens because weak clients in the second partition cannot complete operations as they are partitioned from the current primary. Once they perform the necessary view changes in the second LAN, they resume processing weak operations; this is observed by an increase in the overall throughput of weak operations completed since both partitions can now complete weak operations in parallel – in fact, faster than before the partition due to decreased cryptographic and message overheads and reduced round trip delay of clients in the second partition from the primary in their partition. The duration of the weak operation unavailability in the non-primary partition is proportional to the number of view changes required. In our experiment, since replicas with ids 2 and 3 reside in the second LAN, two view changes were required (to make replica 2 the new primary).

When the partition heals, replicas in the first view detect the existence of concurrency and construct a POD, since replicas in the second LAN are in a higher view (with $v = 2$). At this point, they request a NEWVIEW from the primary of view 2, move to view 2, and then propagate their locally executed weak operations to the primary of view 2. Next, replicas in the first LAN need to fetch the weak operations that completed in the second LAN and needs to complete them before the strong operations can make progress. This results in additional delay before the strong operations can complete, as observed in the figure.

Varying partition duration. Next, we simulate partitions of varying duration as before, for $\alpha = 75\%$. Again, we measure the unavailability of both strong and weak operations using the earlier definition: unavailability is the duration for which the throughput in either partition was less than 10% of average throughput before the failure. With a longer partition duration, the cost of the merge procedure increases since the weak operations from both partitions have to be transferred prior to completing the new client operations.

Figure 6 presents the results. We observe that weak operations experience some unavailability in this scenario, whose duration increases with the length of the partition. The unavailability for weak operations is within 9% of the total time of the partition.

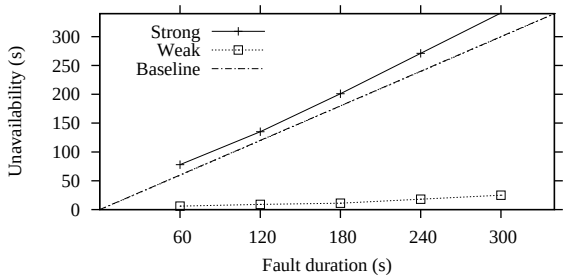

Figure 6: Varying partition durations with concurrent operations. Baseline represents the minimal unavailability expected for strong operations, which is equal to the partition duration.

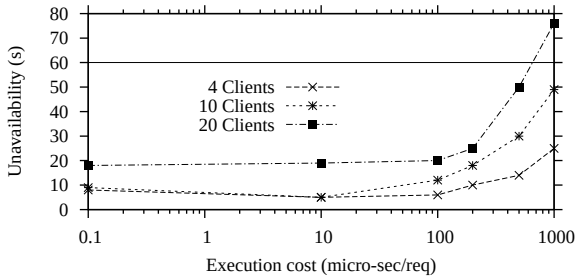

Figure 7: Varying execution cost of operations with increasing request load. 60 second partition duration.

The unavailability of strong operations is at least the duration of the network partition plus the merge cost (similar to that for weak operations). The additional unavailability due to the merge operation is within 14% of the total time of the partition.

Varying execution cost and request load. In this experiment, we vary the execution cost of each operation as well as increase the request load, by increasing the number of clients, to estimate the cost of merges when the system is loaded. For example, the system was operating at peak cpu utilization with 20 clients and operations with 200 μs/operation or more. Here, we set $\alpha = 100\%$. We present results with a partition duration of 60 seconds in Figure 7. We observe that as the cost of operations system load increases, the unavailability of weak operations also goes up. This is expected because the set of weak operations performed in one partition must be re-executed at the replicas in the other partition during the merge procedure. As the client load and the cost of operation execution increases, the time taken to re-execute the operation also increases. In particular, when the system is operating at 100% cpu utilization, the cost of re-executing the operations will take as much as time as the duration of the partition, and therefore the unavailability in these cases is higher than the partition duration. If, however, the system is not operating at peak utilization, the cost of merging is lower than the partition duration.

Varying request size. We ran an experiment with a 5 minute partition, and varying request sizes from 2 Bytes to 1 KB. The results with different request sizes were similar to those shown in Figure 5 so we do not plot them. We observed that increasing the payload size does not significantly affect the merge duration. This is due to the high speed network connection between replicas.

Summary. Our microbenchmark results show that Zeno significantly improves the availability of weak operations and the cost of merging is reasonable as long as the system is not overloaded. This allows Zeno to quickly start processing strong operations soon after partitions heal.

6.2.4 Mix of strong and weak operations

In this experiment, we allow each client to issue a mix of strong and weak operations. Note that as soon as a client issues a strong operation in a partition, it will be blocked until the partition heals. We use a client population of 40 nodes. Each client issues a strong operation with probability p, weak operations with probability $0.8 - p$, and exits from the system with a fixed probability of 0.2. We implement a fixed think time of 10 seconds between operations issued by each client. The think times and the exit probability are obtained from the SpecWeb2005 banking benchmark [11]. Next, we vary p to estimate the impact of failure events such as network partitions on the overall user experience. To give an idea of reference values for p, we looked into the types and frequencies of distinct operations in existing benchmarks. In an e-banking benchmark, and assigning the billing operations to be strong operations, the recommended frequency of such operations follows $p = 0.13$ [11]. In the case of an e-commerce benchmark, if the checkout operation is considered strong while the remaining, such as login, accessing account information and customizations are considered as weak operations, then we obtain $p = 0.05$ [1]. Our experimental results cover these values.

We simulate a partition duration of 60 seconds and calculate the number of clients blocked and the length of time they were blocked during the partition. Figure 8 presents the cumulative distribution function of clients on the *y*-axis and the maximum duration a client was blocked on the *x*-axis. This metric allows us to see how clients were affected by the partition. With Zyzzyva, all clients will be blocked for the entire duration of the partition. However, with Zeno, a large fraction of clients do not observe any wait time and this is because they exit from the system after doing a few weak operations. For example, more than 70% of clients do not observe any wait time as long as the probability of performing a strong operation is less than 15%. In summary, this result shows that Zeno significantly improves the user experience and masks the failure events from being exposed

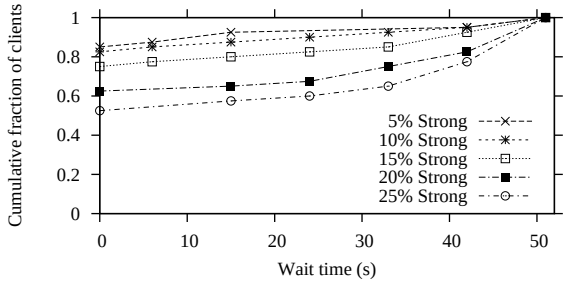

Figure 8: Wait time per client with varying probability p of issuing strong operations.

to the user as long as the workload contains few strong operations.

7 Related Work

The trade-off between consistency, availability and tolerance to network partitions in computing services has become folklore long ago [7].

Most replicated systems are designed to be "strongly" consistent, i.e., provide clients with consistency guarantees that approximate the semantics of a single, correct server, such as single-copy serializability [21] or linearizability [23].

Weaker consistency criteria, which allow for better availability and performance at the expense of letting replicas temporarily diverge and users see inconsistent data, were later proposed in the context of replicated services tolerating crash faults [18, 33, 35, 40]. We improve on this body of work by considering the more challenging Byzantine-failure model, where, for instance, it may not suffice to apply an update at a single replica, since that replica may be malicious and fail to propagate it.

There are many examples of Byzantine-fault tolerant state machine replication protocols, but the vast majority of them were designed to provide linearizable semantics [4, 9, 12, 24]. Similarly, Byzantine-quorum protocols provide other forms of strong consistency, such as safe, regular, or atomic register semantics [30]. We differ from this work by analyzing a new point in the consistency-availability tradeoff, where we favor high availability and performance over strong consistency.

There are very few examples of Byzantine-fault tolerant systems that provide weak consistency.

SUNDR [27] and BFT2F [28] provide similar forms of weak consistency (fork and fork*, respectively) in a client-server system that tolerates Byzantine servers. While SUNDR is designed for an unreplicated service and is meant to minimize the trust placed on that server, BFT2F is a replicated service that tolerates a subset of Byzantine-faulty servers. A system with fork consistency might conceal users' actions from each other, but if it does, users get divided into groups and the members of one group can no longer see any of another group's file system operations.

These two systems propose quite different consistency guarantees from the guarantees provided by Zeno, because the weaker semantics in SUNDR and BFT2F have very different purposes than our own. Whereas we are trying to achieve high availability and good performance with up to f Byzantine faults, the goal in SUNDR and BFT2F is to provide the best possible semantics in the presence of a large fraction of malicious servers. In the case of SUNDR, this means the single server can be malicious, and in the case of BFT2F this means tolerating arbitrary failures of up to $\frac{2}{3}$ of the servers. Thus they associate client signatures with updates such that, when such failures occur, all the malicious servers can do is conceal client updates from other clients. This makes the approach of these systems orthogonal and complementary to our own.

Another example of a system that provides weak consistency in the presence of some Byzantine failures can be found in [34]. However, the system aims at achieving extreme availability but provides almost no guarantees and relies on a trusted node for auditing.

To our knowledge, this paper is the first to consider eventually-consistent Byzantine-fault tolerant generic replicated services.

8 Future Work and Conclusions

In this paper we presented Zeno, a BFT protocol that privileges availability and performance, at the expense of providing weaker semantics than traditional BFT protocols. Yet Zeno provides eventual consistency, which is adequate for many of today's replicated services, e.g., that serve as back-ends for e-commerce websites. Our evaluation of an implementation of Zeno shows it provides better availability than existing BFT protocols, and that overheads are low, even during partitions and merges.

Zeno is only a first step towards liberating highly available but Byzantine-fault tolerant systems from the expensive burden of linearizability. Our eventual consistency may still be too strong for many real applications. For example, the shopping cart application does not necessarily care in what order cart insertions occur, now or eventually; this is probably the case for all operations that are associative and commutative, as well as operations whose effects on system state can easily be reconciled using snapshots (as opposed to merging or totally ordering request histories). Defining required consistency per *operation type* and allowing the replication protocol to relax its overheads for the more "best-effort" kinds of requests could provide significant further bene-

fits in designing high-performance systems that tolerate Byzantine faults.

Acknowledgements

We would like to thank our shepherd, Miguel Castro, the anonymous reviewers, and the members of the MPI-SWS for valuable feedback.

References

[1] TPC-W Benchmark White Paper. http://www.tpc.org/tpcw/TPC-W_Wh.pdf.

[2] Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008.

[3] FaceBook's Cassandra: A Structured Storage System on a P2P Network. http://code.google.com/p/the-cassandra-project/, 2008.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Brighton, United Kingdom, 2005.

[5] Amazon. Discussion Forum: Thread: Massive (500) Internal Server Error. Outage started 35 minutes ago. http://developer.amazonwebservices.com/connect/thread.jspa?threadID=197%14&start=90&tstart=0, 2008.

[6] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Banff, Canada, 2001.

[7] E. Brewer. Towards Robust Distributed Systems (Invited Talk). Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 2000.

[8] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, Jan. 2001. Technical Report MIT/LCS/TR-817.

[9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, New Orleans, LA, USA, 1999.

[10] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[11] S. P. E. Corporation. Specweb2005 release 1.20 banking workload design document. http://www.spec.org/web2005/docs/1.20/design/BankingDesign.html, 2006.

[12] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, 2006.

[13] M. Dahlin, B. B. V. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2), 2003.

[14] J. Dean. Handling large datasets at Google: Current systems and future designs. In Data-Intensive Computing Symposium, Mar. 2008.

[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, 2004.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[17] A. Fekete. Weak consistency conditions for replicated data. Invited talk at 'A 30-year perspective on replication', Nov. 2007.

[18] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[20] Google. App Engine Outage today. http://groups.google.com/group/google-appengine/browse_thread/thread/f7%ce559b3b8b303b?pli=1, 2008.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] J. Hamilton. Internet-Scale Service Efficiency. In *Proceedings of 2nd Large-Scale Distributed Systems and Middleware Workshop (LADIS)*, New York, USA, 2008.

[23] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[25] 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.

[26] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. http://cs.utexas.edu/~kotla/RESEARCH/CODE/ZYZZYVA/, 2008.

[27] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2004.

[28] J. Li and D. Mazières. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[29] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[30] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Symposium on Theory of Computing (STOC)*, El Paso, TX, USA, May 1997.

[31] Netflix Blog. Shipping Delay. http://blog.netflix.com/2008/08/shipping-delay-recap.html, 2008.

[32] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Banff, Canada, 2001.

[33] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), 2005.

[34] M. Spreitzer, M. Theimer, K. Petersen, A. J. Demers, and D. B. Terry. Dealing with server corruption in weakly consistent replicated data systems. *Wireless Networks*, 5(5), 1999.

[35] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Cooper Mountain Resort, Colorado, USA, 1995.

[36] F. Travostino and R. Shoup. eBay's Scalability Odyssey: Growing and Evolving a Large eCommerce Site. In *Proceedings of 2nd Large-Scale Distributed Systems and Middleware Workshop (LADIS)*, New York, USA, 2008.

[37] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, Boston, MA, USA, 2002.

[38] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. In *Proceedings of ACM Symposium on Operating*

System Principles (SOSP), Stevenson, WA, USA, 2007.

[39] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[40] H. Yu and A. Vahdat. Design and Evaluation of a Conit-Based Continuous Consistentcy Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3), 2002.

A Non-PKI based Zeno

In this section, we describe a version of Zeno that uses MACs in the normal case operation (i.e., for authenticating REQUEST, REPLY, ORDERREQ, COMMIT, and CHECKPOINT messages) but continues to use PKI based signatures for the view change messages. Using MACs in the normal case operation affects the request processing, conflict detection, merge, and view change logic. We describe these modifications in this section.

A.1 Request validity proof

A common problem in MAC based protocols is verifying client requests. For example, a faulty client can carefully construct an authenticator that only verifies at the primary replica. Such a request will be assigned a sequence number by the correct primary, but will not be verified by other replicas and hence will be dropped. Unfortunately, it is impossible to detect if the corrupted authenticator is due to a faulty primary or a faulty client. Existing protocols provide mechanisms to deal with such requests. At a high level, these mechanisms ensure that all non-faulty replicas either pick the same request for execution at a sequence number or all decide to choose a special *no-op* operation. Since Zeno is based on Zyzzyva, we use the mechanism proposed by Zyzzyva [25].

Briefly, the mechanism works as follows. If a replica is unable to verify a client request, it requests an authentication proof from the primary. The primary responds with a proof, which can be one of the following:

1. same request signed using PKI by the client
2. a validity proof, consisting of PKI signatures from $f+1$ replicas that have correctly verified the client request
3. an invalidity proof, consisting of PKI signatures from $2f+1$ replicas, out of which only f or less replicas correctly verified the client request

If the primary does not possess this proof, the primary asks all replicas to check if they can verify the client request. If a replica can correctly verify the request, it sends a PKI signed message to the primary to confirm the validity of the request, otherwise it responds with an "authentication failure" message, also signed with PKI. The primary receives these responses, constructs the proof, and sends the proof to all replicas. All correct replicas make identical decisions based on the proof sent by the primary. If the primary is faulty and sends two different authentication proofs, other replicas initiate a view change.

Using MACs in the normal case also affects the specification of our service in the following way. Suppose a faulty client sends a request to a weak partition with a corrupted authenticator such that the request verifies only at $(f+1)$ replicas, out of which only one is non-faulty. That correct replica will be responsible for forwarding the request to other replicas during future merges. However, since the authenticator is corrupted, the request will not verify at other non-faulty replicas, causing it to be dropped by Zeno during subsequent merges. Therefore, it is possible for requests from faulty clients to appear in the tentative history but to disappear from the linearized history of the service. In contrast, requests from correct clients will always appear in the linearized history of the service. In order to handle this case, we modified the merge operation in the specification. (Figure 9).

Transitions:

MERGE
Pre: $|H| \geq 2$
Eff: select $\{h, h'\} \subseteq H$
$h'' :=$ merge h and h' - o_f
remove duplicates
o_f are subset of all operations from faulty clients
$committed(h'') := \max(committed(h), committed(h'))$
$H := H - \{h, h'\} + \{h''\}$

Figure 9: Modified Merge procedure for MAC-based Zeno. o_f represents the subset of operations issued by faulty clients.

Similarly, the liveness property of Zeno is modified as follows:

(L2') If there exists an eventually synchronous set of $2f+1$ correct servers Π', then every weakly complete request issued by a *correct client* or a strong request issued by a correct client is eventually committed.

Recall that in the PKI-based Zeno, every weakly completed request from faulty clients were also guaranteed to be eventually committed.

A.2 Reply and Commit Messages

We now describe the modification to how replicas construct reply and commit messages. First, backup replicas do not include the OR in REPLY messages. Second, backup replicas also do not include the OR in the COMMIT message. Both are due to inherent weakness

of MACs—even if a replica or a client detects misbehavior of the primary, others may not be able to verify it. These modifications are identical to how Zyzzyva operates when using MACs.

A.3 Conflict Detection

The conflict detection procedure in the PKI version had three subcases: (i) Proof of Misbehavior (POM), (ii) Proof of Divergence (POD), and (iii) Proof of Absence (POA). Below, we consider the implication of using MACs on these subcases and show that POM/POD/POA do not guarantee that conflicts will be detected.

POM/POD A POM (POD) essentially consists of two OR messages for the same sequence number that are sent by the same (different) primary and that have been assigned different requests. With PKI, if one non-faulty replica collects a POM (POD), it can convince other non-faulty replicas of the primary's misbehavior and trigger a merge. Unfortunately, MACs do not provide this capability since an OR that conflicts with the local OR stored at a replica may not verify correctly at the replica. Therefore, a faulty primary can keep non-faulty replicas from detecting divergence and no POM (POD) is generated by any non-faulty replica. Even if a non-faulty replica is able to generate a POM (POD), we can not guarantee that other non-faulty replicas will generate the identical POM (POD) since the conflicting OR's may not verify at *all* non-faulty replicas.

POA Since a POA contains the ORs from an earlier view that spans the ORs contained in the NEWVIEW message, the ORs in the POA may not verify correctly at other non-faulty replicas. Therefore, POAs also share the same problem as POM/PODs.

A.3.1 MAC-based conflict detection

We now present a new conflict resolution procedure that works with MACs.

Overview In traditional BFT protocols, if replicas are unable to commit requests, they can trigger a view change. This is important for detecting a faulty primary that sends either a divergent history in ORs or sends ORs that do not correctly verify at correct replicas. By checking that at least $2f+1$ replicas have verified and received identical history in the ORs, existing protocols ensure that a faulty primary that sends corrupted MACs or inconsistent ORs is eventually replaced. At a high level, we use a similar idea to detect conflicts. In a strong partition, replicas check if they can commit requests. Otherwise, non-faulty replicas suspect divergence and initiate a view change. In a weak partition, replicas check if the history in the OR they received from the primary matches the history present in the OR received by *all* other replicas in the partition. Otherwise, non-faulty replicas suspect divergence and initiate a view change.

Divergence in the same view Suppose replica i in view v_i receives a message m from replica j for sequence number n and view v_j. Here, m is either an OR or a COMMIT.

Algorithm 1 presents the logic for detecting conflicts within the same view, i.e., $v_i = v_j$.

Each replica maintains two two-dimensional arrays, `matchingHistory` and `divergentHistory`, to detect conflicts. We first describe how these arrays are initialized. When a replica receives an OR from the primary with id k for n, it stores the OR in `matchingHistory[n][k]`.

If a replica expects to commit at n, e.g., for generating a checkpoint at n or when the request assigned to n is a strong operation, then the $\delta[n]$ timer is started. When a replica i sends a COMMIT message, it stores the COMMIT message in `matchingHistory[n][i]`. When a replica receives a COMMIT message from replica j with a history chain digest identical to the history chain digest it has received from the primary in the OR for n, the replica stores the message in `matchingHistory[n][j]`. If an OR sent by the same primary but with a different history is received, then the primary is faulty, and replica initiates a view change. If a COMMIT is received from another replica with a different history, then the replica stores the message in `divergentHistory[n][j]`.

When the $\delta[n]$ timer expires, replicas calculate the size of these arrays. Let |`matchingHistory[n]`|=M, meaning that M replicas share a matching history for sequence number n. Let |`divergentHistory[n]`|=D, meaning that D replicas have divergent histories for n. Let $C = M + D$, which represents the total number of other replicas a local replica is able to communicate with within δ time period. Then, the replica checks that:

1. $M \geq (2f+1)$ // *which implies* $C \geq (2f+1)$
2. $(C < 2f+1) \wedge (D == 0)$

If either of these conditions is satisfied, the replica concludes that there is no divergence. Otherwise, the replica suspects divergence and initiates a view change.

A careful reader will ask: why do we have the δ timer and the condition on M and D in the MAC-version of the protocol, compared to conflict detection in the PKI version, where simple comparison of history chain digest was enough? The reason is intimately tied to the weaker properties of MACs compared to PKI signatures. We rely on our synchrony assumption to ensure that, within the δ time period, every pair of non-faulty replicas can communicate with each other. If there is divergence in the

Algorithm 1 Divergence in the same view ($v_i = v_j$)

```
1:  // m is either an OR or a COMMIT message
2:  if m is not properly authenticated then
3:      Return.
4:  // Let remote replica's id be j
5:  // Assume i's highest sequence number is n_highest
6:  if n > n_highest then
7:      Multicast fillHole for [n_highest + 1, n]. Return.
8:  else
9:      // We must have the ORDERREQ for n. Call it m_i
10:     if m is OR then
11:         // Compare histories to detect divergence
12:         if m.h ≠ m_i.h then
13:             /* History is divergent. Both OR's are sent
                by same primary, so replace it.*/
14:             Initiate view change. Return.
15:     if m is COMMIT then
16:         if m.h ≠ m_i.h then
17:             /* History is divergent. OR and COMMIT are
                sent by different replicas. Therefore, does not
                prove that primary is faulty.*/
18:             Store m in divergentHistory[n][j].
19:         else
20:             // History matches
21:             Store m in matchingHistory[n][j].
```

history of non-faulty replicas, neither of the above conditions will hold, causing a view change to be initiated, and hence a merge of the divergent histories.

We now look at these conditions in detail:

1. Assume that all non-faulty ($2f+1$ or more) replicas in a partition can eventually communicate with each other in δ time. For committing requests, the $2f$ COMMIT messages sent by replicas must match the OR that the primary sent, for a total of $2f+1$ replicas with matching history. Otherwise, requests can not be committed because of divergence in the history of non-faulty replicas and view change must be initiated. Therefore, $M \geq 2f+1$ and hence $C \geq 2f+1$.

2. Assume that fewer than $2f+1$ non-faulty replicas can eventually communicate with each other in δ time (a weak partition). To guarantee progress for weak operations, we require at least $f+1$ non-faulty replicas. However, if the primary is faulty, it can send divergent histories in the OR's to different replicas, which will cause the reply from $f+1$ non-faulty replicas to not match, and therefore prevent the weak operations from completing at clients.

   By requiring $D == 0$, we require that all reachable replicas have identical history. In a synchronous weak partition, this condition ensures that divergence introduced by a faulty primary will be detected and view change initiated.

   Note that it is also necessary to require $D == 0$. It is not sufficient to have $M \geq f+1$ in a weak partition ($C < 2f+1$). To see this, consider the case where out of $f+1$ replicas that have matching histories, f are faulty. These faulty replicas may not respond, preventing the client from receiving $f+1$ matching replies, which violates our liveness requirement **L1**. Ensuring that no divergence exists among replicas during a weak partition prevents this problem.

   However, requiring identical history at all replicas introduces a potential liveness problem. Suppose a faulty replica sends a COMMIT with a history that is divergent from the history sent by the correct primary. The non-faulty replicas will suspect the correct primary to be faulty and trigger a view change, even when the primary is correct. The view change affects performance but does not violate the liveness guarantee since Zeno makes progress in between view changes (Section A.4).

Divergence across views So far we have presented the conflict detection procedure for replicas in the same view. Now consider the case where replicas are in different views (Algorithm 2). When replica i receives a message m from replica j, it checks if the remote view in m is higher than the local view. If so, i requests the NEWVIEW from j. If the local view is higher, on the other hand, then i sends the local NEWVIEW to j. When a replica receives the NEWVIEW message for a higher view, it verifies the message, and then moves to the higher view. At this point, the replica removes all previous state for conflict detection; i.e, it clears `matchingHistory[][]` and `divergentHistory[][]`, cancels the $\delta[]$ timers, and proceeds according to the merge procedure described in Section A.5. Note that it is safe to discard these arrays and timers since, once $f+1$ non-faulty replicas are synchronous in the same view, any divergence will be detected using the procedure for detecting divergence in the same view.

Algorithm 2 Divergence across views ($v_i \neq v_j$)

```
1:  // m is either an OR or a COMMIT message
2:  if m is not properly authenticated then
3:      Return.
4:  if v_j > v_i then
5:      Ask for NEWVIEW message. Return.
6:  if v_j < v_i then
7:      Send our NEWVIEW message. Return.
```

A.3.2 Impact on safety

The safety guarantee of Zeno for strong operation still holds since the conflict detection protocol described above does not change how replicas commit operations.

The argument of safety for weak operations directly follows the argument in the PKI version. The weak operations still require at least $f+1$ matching replies. A non-faulty replica does not maintain two concurrent histories, therefore, at most $maxhist = \lfloor \frac{n-|f'|}{f+1-|f'|} \rfloor$ concurrent histories can exist in the presence of f' faulty replicas.

A.3.3 Impact on liveness

With respect to liveness, we need to argue that, when a faulty primary introduces conflicts, eventually, the conflicts are detected and view change is initiated. Also, we need to argue that progress is made when the primary is non-faulty.

Same view Assume that all replicas are in the same view. We need to ensure that: (i) view changes are initiated for the faulty primary in a strong partition, (ii) view changes are initiated for the faulty primary in a weak partition, and (iii) progress is made when the primary is non-faulty.

1. Assume an eventually synchronous strong partition with at least $2f+1$ non-faulty replicas and a faulty primary. If replicas are unable to commit operations, then $M < 2f+1$, since otherwise replicas will be able to commit requests. Hence, if there is divergence, correct replicas will initiate a view change.

2. Assume an eventually synchronous weak partition with at least $f+1$ correct replicas and a faulty primary. Suppose a correct client does not receive $f+1$ matching replies. When replicas exchange the COMMIT messages (e.g., during the checkpoint interval), correct replicas will obtain at least one COMMIT message that does not match the history it received in the OR from the primary. This ensures that $D > 0$, causing all correct replicas in the weak partition to initiate a view change.

3. Assume an eventually synchronous weak partition with at least $f+1$ correct replicas, the primary is correct, and at least one replica is faulty. A faulty replica can send a COMMIT message with history that diverges with the history sent by the correct primary. This will cause D to be greater than 0, triggering the non-faulty replicas to initiate a weak view change. However, replicas give priority to completing ORs in the same view before processing the view change messages (see Section 4.5). Therefore, even though faulty replicas can cause continuous view changes in a weak partition, Zeno makes progress as long as $f+1$ non-faulty replicas are synchronous and hence maintains its liveness guarantee.

Different views We have so far seen that a faulty primary will be eventually replaced as long as enough non-faulty replicas are in the same view and synchronous. Now consider the case where replicas are in different views. In this case, replicas from the lower view will request the NEWVIEW message but will continue to operate in their current view. Once a NEWVIEW message is received, a replica can immediately check its validity (since it is signed using PKI), and move to the higher view. If there are at least $f+1$ non-faulty replicas in a synchronous partition, each non-faulty replica will move to the highest view of any given non-faulty replica. At that point, if the primary is correct, the weak operations can complete or divergence will be detected and a view change triggered.

A.4 View change

Since Zeno is based on Zyzzyva, we first describe Zyzzyva's view change protocol when MACs are used in the normal case. Note that the view change protocol messages are signed using PKI signatures.

A.4.1 View Change in Zyzzyva

When a replica suspects the current primary to be faulty, it sends a IHATETHEPRIMARY message to all other replicas. Once replica i receives $f+1$ IHATETHEPRIMARY messages, it gathers a proof for its local commit certificate (CC) and checkpoint certificate by requesting all other replicas to send their signature for the highest CC and the checkpoint that replica i locally possesses. If replica j has received an order request for the specified CC, then it replies with a signature for that order request. If replica j has sent the checkpoint message previously then j responds with a signature for that checkpoint. Replica i considers $f+1$ signatures to complete the commit and the checkpoint certificate proofs. Once replica i has these proofs, it sends a view change message to the new primary.

The new primary waits to receive $2f+1$ *non-conflicting* view change messages. Two view change messages are conflicting if they contain CC proofs for different requests at the same sequence number. Since CC proposed by non-faulty replicas do not conflict, and if the wait time is sufficiently long, a non-faulty primary will obtain $2f+1$ non-conflicting view change messages.

The new primary then constructs a NEWVIEW message based on the $2f+1$ non-conflicting view change messages and sends it to other replicas.

A.4.2 Challenge for Zeno

For strong partitions, Zeno keeps the structure of the view change similar to Zyzzyva. However, view changes in weak partitions are challenging. Recall that a non-faulty replica waits to receive at least $f+1$ proofs (including its own) for its CC and the checkpoint before sending a VIEWCHANGE message. This poses a challenge for weak partitions with less than $2f+1$ non-faulty replicas. In this situation, a replica with the highest CC may not obtain a proof since remaining non-faulty replicas in the partition may not have received the ordering corresponding to the CC and other replicas could be faulty and not respond. Therefore, the new primary may not get enough view change messages to construct the NEWVIEW message, compromising the liveness of the view change protocol.

Solution The eventual consistency guarantee provided by Zeno in a weak partition offers an opportunity to design a new view change protocol that is live. We introduce the following three changes in the view change logic described earlier.

1. Like Zyzzyva, before sending the VIEWCHANGE message, a replica requests a proof of its local CC and checkpoint message from other replicas. However, unlike Zyzzyva, a replica starts a timer for δ time after sending the request for the proof. A replica then sends the view change message if either it has received the proof or if the δ timer expired (even if has not yet received the requested proofs). This relaxation is crucial for view change to be live and is sufficient for eventual consistency semantics, as described below in Section A.4.3.

2. The new primary selects the $f+1$ non-conflicting view change messages to construct the NEWVIEW message. (This is based on the observation that view change messages sent by non-faulty replicas do not conflict.) Hence, eventually, a new primary will be able to send a NEWVIEW message. Note that there may not be a proof of the highest CC and checkpoint message in the NEWVIEW message.

3. A replica who possesses a CC that is conflicting with the CC picked in the new weak view does not participate in the weak view and continues to send view change messages for the next view. This ensures that committed operations are not lost across view changes.

Dealing with strong operations in a weak partition In a weak partition, strong operations can not be completed. A client will therefore keep retransmitting the strong operation. In a traditional BFT protocol, such retransmissions will lead to continuous view changes. In Zeno, however, we must avoid these view changes since weak operations must complete efficiently even in a weak partition. When a strong operation is received and ordered by the primary, replicas start the commit phase by sending a COMMIT message. The conflict detection mechanism described earlier is also triggered whenever replicas intend to commit an operation. If a conflict is detected, view change is triggered. Otherwise, if no conflict is detected, the retransmission is neglected and unnecessary view changes are avoided.

However, if the operation is not yet ordered, the usual logic is followed, i.e., a replica forwards the operation to the primary and starts the IHATETHEPRIMARY timer. If ordering is not received before the timer expires, a replica sends the IHATETHEPRIMARY message to all replicas.

A.4.3 Impact on safety

We argue that the safety of strong operations is not affected by weak view changes. The reason is that a non-faulty replica never replaces its own CC with a conflicting CC. Therefore, the correct replicas that participated in a commit operation ensure that the order of the committed operation does not change across weak view changes.

The argument of safety for weak operations directly follows the argument in the PKI version. The weak operations still require at least $f+1$ matching replies. A non-faulty replica does not maintain two concurrent histories, therefore, at most $maxhist = \lfloor \frac{n-|f'|}{f+1-|f'|} \rfloor$ concurrent histories in the presence of f' faulty replicas.

A.4.4 Impact on liveness

Suppose a client is unable to complete its weak operations. If the primary is faulty and is introducing conflicts, the conflict detection procedure described in Section A.3.1 will eventually detect the conflict and initiate a view change. If the primary is dropping requests or is silent, traditional mechanisms will initiate a view change. Now we argue that the weak view change protocol is live.

Assume an eventually synchronous weak partition and that a non-faulty replica becomes the primary for the next view. Each of the non-faulty replicas will either receive the proof of their CC and checkpoint or timeout and sends the VIEWCHANGE message to the primary. Therefore, the new primary will be able to receive at least $f+1$ view change messages that do not conflict with each other and sends the NEWVIEW message. (Recall that the view change message sent by non-faulty replicas do not conflict with each other.)

Assume an eventually synchronous weak partition and that a faulty replica becomes the primary and sends a NEWVIEW message that conflicts with the highest CC

of a correct replica. That correct replica will not participate in the new view. If the view is live, i.e., the faulty primary assigns consistent ordering to weak operations, the weak operations will be completed. (Although, it is possible that the highest CC is not incorporated in this view.) Otherwise, more view changes will be triggered and eventually a non-faulty replica will be elected as the primary.

We now argue that, in an eventually synchronous weak partition, retransmissions of strong operations by correct clients can not cause continuous view changes if none of the replicas are faulty. (Recall that a faulty replica can cause continuous view changes in a weak partition, as noted above, but progress is made in-between view changes.) If none of the replicas are faulty, the conflict detection at the commit time (e.g., for checkpoint interval) will find that $D = 0$. This in turn ensures that none of the replicas initiates a view change and the retransmission of the strong operation is neglected.

Finally, we argue that while a faulty replica in a weak partition can trigger continuous view changes, Zeno makes progress in between such view changes. The argument is similar to Section A.3.3.

A.5 Merge

Here, we describe how replicas merge operations that they have weakly completed but not yet committed, for example the operations that were completed in a weak partition. Each non-faulty replica maintains a buffer, `tentative_req`, of client requests that it correctly verified but that have not yet committed. Whenever a replica moves to a higher view, either by participating in a view change protocol or via obtaining a NEWVIEW message, it sends the requests in `tentative_req` buffer to all replicas. Once a request is committed or dropped (when a proof of its unauthenticability is obtained, see Section A.1), the replica removes the request from the `tentative_req` buffer.

Liveness of weak operations Now we argue that Zeno preserves the liveness of weak operations (**L2'**). Recall that for eventual consistency, each weakly complete operation from a non-faulty client must get committed eventually. The merge procedure described above ensures that every weakly completed operation is forwarded to all correct replicas. Since the request is sent by a correct client, it has a correct authenticator, and therefore it will be correctly verified at other non-faulty replicas. Replicas will wait for a timeout and then forward the request to the current primary if the request is not already ordered. The checkpoint protocol is periodically initiated, ensuring that operations are committed. If the current primary is faulty, it will be replaced eventually by a correct primary as per the conflict detection and view change protocol. Once a non-faulty replica is elected as the primary, the commits will succeed and weak operations will appear in the committed history.