Stronger Abstractions and Performance Guarantees for Building Strongly Consistent Distributed Services

Christopher Charles Hodsdon

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Advisers: Professor Wyatt Andrew Lloyd

and Professor Ethan Benjamin Katz-Bassett

May 2023

Abstract

Strongly consistent distributed services require complex coordination among the services' machines. Building these services thus requires designing and reasoning about complex coordination. To make building strongly consistent distributed services easier, designers build services atop abstractions that simplify reasoning about the service. Two abstractions used today are the multi-sequence abstraction and replicated state machines.

The multi-sequence abstraction simplifies ordering operations on sharded data; it explicitly orders the operations on each shard it accesses. Existing implementations have two shortcomings: failures can result in some multi-sequence numbers never being assigned, exposing a noncontiguous multi-sequence, which requires designing and implementing complex coordination in the form of consensus to handle, and existing implementations do not scale.

The replicated state machine (RSM) abstraction is used by many of today's services to ensure correctness and resilience despite failures. It provides the abstraction of "a single machine that does not fail." However, with RSM protocols used today, an individual slow replica can slow the entire RSM. RSMs are commonly geo-distributed, but current state-of-the-art protocols that provide the stronger slowdown-tolerant abstraction do not have comparable latency to the state-of-practice in a wide-area network or are unable to handle certain types of slowdowns.

This dissertation shows that it is possible to build systems that strengthen both abstractions while providing performance guarantees.

First, we posit that sequencers should expose our new contiguous multi-sequence abstraction. Contiguity guarantees every sequence number is assigned an operation, simplifying and strengthening the abstraction. Without scalability in the implementation of the contiguous multi-sequence, a service would need to be redesigned once it has outgrown the implementation. We design and implement MASON, the first system to expose the contiguous multi-sequence abstraction and the first to provide a scalable multi-sequence. MASON is

thus an ideal building block for consistent, scalable services. Our evaluation shows MASON unlocks scalable throughput for two strongly consistent services built on it.

Second, we seek to provide the stronger RSM abstraction of "a single machine that does not slow down" with AVICENNA, a slowdown-tolerant replicated state machine protocol that has comparable latency to the current state-of-practice in the wide-area.

Acknowledgements

I thank my advisers Wyatt Lloyd and Ethan Katz-Bassett whose support and dedication throughout my graduate studies helped me to grow as a researcher. Wyatt stoked my interest in distributed systems during an undergraduate internship at the University of Southern California and in working with Wyatt and Ethan my interest in distributed systems and systems in general continued to grow. Among many other lessons, they taught me the importance of communication, of viewing problems at a high level before getting lost in the weeds, of being precise, and of always asking the question that was tacked to the wall above Wyatt's desk: "Why?". Ethan's attention to detail and enthusiasm for every aspect of research continuously improved how I work and the quality of the work itself. Wyatt's positivity always kept me going even when times were rough.

I thank Siddhartha Sen, who's advice and encouragement through the final two years of my graduate studies helped to shape my work and make it the best that it could be. There were multiple occasions in the lab at Microsoft Research where we brainstormed ideas in a conference room or thought through problems over lunch, and these sessions helped me to learn how to think through problems and find practical solutions.

I would also like to thank the other members of my dissertation committee, Ravi Netravali and Michael Freedman, whose feedback improved this work.

I was extremely fortunate to have met Tejas and Rajiv Gandhi who became my mentor during my undergraduate studies, and without whom I would never have begun down this path. I have learned many invaluable lessons from Rajiv who initiated my academic interest in computer science and frequently said "Work hard and good things will happen", which I repeated to myself many times during my studies.

I thank Theano Stavrinos who became a supportive friend and collaborator. Theano collaborated on a large portion of the work that went into this dissertation and was invaluable in bouncing ideas off of each other and helping with writing, implementation, and whatever other obstacles came our way.

During my time in the labs at USC and Princeton University I met many supportive friends and peers including: Khiem Ngo, Haonan Lu, Jennifer Lam, Andrew Or, Zhenyu Song, Yue Tan, Brandon Schlinker, Yi-Ching Chiu, Mohsin Ali, Luis Pedrosa, Jianan Lu, Ashwini Raina, Sam Ginzburg, Nanqinqin Li, Shai Caspin, Jeffrey Helt, Natalie Popescu, David Liu, Anja Kalaba, and Neil Agarwal, along with others too many to list. I began research in distributed systems working with Khiem, Haonan, and Wyatt on the SNOW project and Khiem and Haonan quickly became my friends who both helped me to think about systems and research. Khiem deserves special mention for being my lunch buddy, having coffee chats, and gracefully answering my many questions. Mohsin Ali, apart from being a delightful person to work with and learn from, deserves recognition for contributing to early stages of the work on sequencing. Neil Agarwal was always nice and helped to motivate me. Jennifer Lam became a good friend, was always supportive, and would give constant encouragement.

I would like to thank the faculty that fostered supportive and vibrant lab environments: Ramesh Govindan, Minlan Yu, Wyatt, and Ethan at USC and Amit Levy, Ravi Netravali, Michael Freedman, Kai Li, Jennifer Rexford, and Wyatt at Princeton. I thank Nicki Mahler for always being helpful and responsive.

I would also like to thank Asaf Cidon, his students, and Ethan's students for welcoming me with open arms during my short time at Columbia University.

I would like to thank Mark Lippincott, who I met while attending Rajiv Gandhi's summer program, for being my friend, showing support, driving us way too many times back and forth between Princeton and Rutgers–Camden during our undergraduate studies, and patiently waiting as I delayed such rides.

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1564242, CNS-1835253, CNS-1910390, and CNS-1827977, and a gift from Comcast. Any opinions, findings, and conclusions or recommendations expressed

in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Comcast.

Finally, I would like to thank my family especially my grandparents Blanche and Charles O'Neill, my parents Diane O'Neill Hodsdon and Craig Walter Hodsdon, and my brother Craig Freeman Hodsdon for their unconditional love and support.

To my family

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most of today's networked services cannot run on a single machine. Single machines can store a limited amount of data, can provide a limited amount of throughput, and can fail, potentially causing data loss. To overcome these limitations services must be distributed across many machines to store more data, provide more throughput, and to store multiple copies of the same data across multiple servers to prevent data loss. To make it easier for applications to reason about concurrent accesses to this data many distributed services aim to provide strong consistency, which allows applications to view the service as a single machine. Designing and building strongly consistent distributed services is a complex task that requires domain knowledge and expertise; it requires coordination among machines to consistently order and execute operations.

To reduce design complexity designers of distributed services use abstractions provided by other services as building blocks to simplify building their own service. Two of the most widely used class of abstractions provides consistent ordering of operations across distributed servers and a machine that executes those operations without failing. An abstraction that provides consistent ordering of operations is the *multi-sequence* abstraction which simplifies reasoning about the order of concurrent operations within and across subsets of data, *shards*. An abstraction that executes operations without failing is the replicated state machine

(RSM) abstraction. RSMs are commonly used as building blocks for shards to provide the abstraction of a non-failing single machine despite a specified number of failures. Current or common implementations of these abstractions can be bottlenecked by a single-machine either preventing scalability or hurting latency when a single machine is slow.

Previous work defined a *sequence* abstraction that globally orders all operations [5, 62]. More recent work, and this dissertation, target the multi-sequence abstraction that only explicitly orders operations when part of the operation executes on the same shard [44, 71]. This allows operations spanning multiple shards to only be ordered with respect to other operations on intersecting shards, reducing contention compared to ordering all operations globally, and improving throughput and latency.

The multi-sequence abstraction uses a collection of *sequence spaces*, i.e., logically independent sequences of strictly increasing integers, to provide a strictly serializable ordering of accesses to different shards of the service's data. An operation that needs cross-shard ordering gets an atomically assigned *multi-sequence number* containing a sequence number from the sequence space of each shard the operation accesses. An *execution protocol*, designed by the service developer, defines the sequence spaces involved in an operation and how shards use multi-sequence numbers to execute operations. Driven by the execution protocol, the service's servers use the sequence numbers to order operations on the shard(s) they manage, with the multi-sequence numbers atomically ordering operations relative to other operations to provide strong consistency. Operations ordered by multi-sequence numbers can be executed without coordination across servers, making it easier to build strongly consistent, scalable, and efficient services.

However, the abstraction used by recent services is a *noncontiguous multi-sequence*: failures can cause *holes* in the sequence space, i.e., sequence numbers that are never used. To preserve consistency, a service must identify and reason about all holes. Identifying holes requires service-wide coordination between the service's servers to reach consensus on whether a sequence number has an associated operation that can be recovered. If not, then it

is a hole, and the servers must coordinate to avoid using any sequence numbers that are part of the same multi-sequence number as the hole. Implementing consensus and service-wide coordination to handle holes significantly complicates execution protocol design (§2.2).

In addition to being noncontiguous, existing implementations of the multi-sequence abstraction [44, 71] suffer from a second limitation: they have an ordering throughput ceiling that limits the throughput of any services built on top of them. These implementations use a *monolithic sequencer*, a single machine whose only task is to hand out multi-sequence numbers, enabling low-latency ordering that is easy to reason about. A monolithic sequencer can order operations with higher throughput than coordination-based mechanisms, but this design can *only achieve ordering throughput up to the throughput limit of a single machine.* Thus, a service built on a monolithic sequencer cannot scale; it is bottlenecked by a single machine.

In addition to ordering operations across shards, strongly consistent distributed services need to ensure they do not lose data. To manage multiple copies of data while ensuring operations are executed in the same order across the copies, many distributed services use replicated state machines (RSMs) as an important building block. An RSM is a set of replicas (individual processes) that use consensus to agree on an ordering for client commands to ensure correctness and resiliency in the case of machine failures, providing the abstraction of a single non-failing machine in the face of up to a certain number of simultaneous failures [65]. RSMs are used extensively, for example for coordination services like Chubby [8] or ZooKeeper [26] or for distributed databases like Spanner [12].

While an RSM provides fault tolerance, with common RSM designs a single slow machine can slow the progress of the entire set of replicas. For example, in leader-based protocols, if the leader becomes slow, the RSM will be as slow as the leader. Slowdowns can occur for many reasons: network congestion can delay messages, hardware bugs can cause hardware to perform slower but still correctly (e.g., corruption detection and correction), and software can have pauses during normal execution (e.g., language garbage collection) [22].

If an RSM becomes slow and delays a response to an application, the application cannot tell the difference between it being slow or failed. Thus, an RSM that can become slow weakens the abstraction. Beyond weakening the abstraction, RSMs becoming slow is undesirable for myriad reasons: it affects tail latency, causes unpredictability in performance, makes it hard to achieve ambitious availability targets, and could ripple and affect other services that rely on the RSM.

RSMs are often geodistributed across a wide-area network (WAN) so that replicas are located closer to clients providing lower latency, and so correlated replica failures are uncommon. Designing a protocol and tolerating slowdowns in the wide-area is challenging because latency between replicas is high (on the order of tens to hundreds of milliseconds versus microseconds in a datacenter), heterogeneous (some replicas are closer to each other than other replicas), and highly variable (wide-area latency can vary drastically over time).

Thesis statement: *It is possible to build stronger abstractions for building strongly consistent distributed services that are not bottlenecked by a single machine.*

In defense of this statement this dissertation introduces a new abstraction and systems that overcome the limitations of single machines. We introduce the *contiguous multi-sequence* abstraction and a system, MASON, that implements the abstraction while preventing the single-machine sequencer from becoming a throughput bottleneck. We also examine slowdown tolerance for replicated state machines in the wide-area for the first time and present AVICENNA a slowdown tolerant RSM designed for the wide-area. MASON and AVICENNA both use the idea of having servers proxy operations on behalf of other replicas: MASON for fault tolerance of operations and AVICENNA for faster reaction of slowdowns.

The contiguous multi-sequence abstraction assigns exactly one operation to every integer in each sequence space such that no sequence space has a hole. Contiguity strengthens the multi-sequence abstraction over its existing noncontiguous counterpart by hiding consensus

and service-wide coordination, simplifying the development of services. Some existing services use the noncontiguous multi-sequence abstraction internally to expose higher-level abstractions like distributed databases [44, 71]. Compared to higher-level abstractions, the contiguous multi-sequence supports developing more diverse functionality, e.g., ephemeral objects (§3.4).

Our system, MASON, addresses the ordering throughput limitation. MASON is a building block for distributed services that provides the contiguous multi-sequence abstraction with no ceiling on ordering throughput, unlocking scalability for services that were previously unscalable. MASON's contiguous multi-sequence implementation enables services to (1) use simple execution protocols that need not incorporate consensus or service-wide coordination and (2) scale to achieve service throughput far higher than what is possible with monolithic sequencers.

Our key insight is that MASON can enable simple execution protocols and scalability via a layer of replicated proxies between clients and a monolithic sequencer. To overcome the failure modes that expose holes, the proxy layer provides fault tolerance for clients and the sequencer, guaranteeing the contiguous multi-sequence abstraction. To overcome the monolithic sequencer's ceiling on ordering throughput, proxies batch requests for multi-sequence numbers. This batching is perfect, in that the sequencer does no more work to allocate one million contiguous numbers than it does to allocate a single number. Each replicated proxy operates essentially independently, allowing the proxy layer to scale out; adding more proxies increases ordering throughput. These techniques enable MASON to scale: if the sequencer is the bottleneck, proxies increase batch size; if the proxy layer is the bottleneck, more proxies are added.

Our evaluation shows MASON provides scalable ordering throughput: with one sequence space, MASON achieves ~16.7 Mops/sec with 24 proxy machines, scaling to ~31.5 Mops/sec with 48 proxy machines. MASON's tradeoff for a stronger abstraction and scalable ordering throughput is higher latency relative to monolithic-sequencer designs, since the proxies and

a single round of replication are on path for each request. MASON's latency is still low, however, with a median latency of ~243 μs at the reported throughputs.

We demonstrate MASON's value as a building block by using it to implement Corfu-MASON, a distributed shared log modeled after CORFU [5]; and ZK-MASON, a distributed prototype of the coordination service ZooKeeper [26]. With MASON's strong abstraction, it was easy to build these services that consistently execute cross-shard operations (§3.4). MASON also unlocked scalability for them in contrast to their fundamentally unscalable original designs. Specifically, our implementation of CORFU's original design is limited to ~14.1 Mops/s (nearly line rate for a sequencer with a 10G NIC, ~14.5 Mops/s). Building it on MASON lets it scale from ~7.3 Mops/s (one server) to ~29.1 Mops/s (four servers). Our implementation of ZooKeeper's original design is limited to ~150 Kops/s; its MASON-based implementation scales from ~1.3 Mops/s (one server) to ~7 Mops/s (eight servers).

This dissertation also examines slowdown tolerance in the wide area for the first time. It addresses two problems: First, the definition of slowdown tolerance in recent work makes it unachievable in the wide area and difficult to compare slowdown tolerance across protocols [55]; we redefine slowdown tolerance to apply to both a wide-area and local-area network in a way that enables comparison across protocols (§4). Second, there is no protocol that correctly reacts to slowdowns and that has comparable latency to the state-of-practice (Multi-Paxos and other leader-based protocols) in a wide-area network.

We present AVICENNA, the first RSM protocol to correctly tolerate the slowdown of any one replica that has latency comparable to the state-of-practice in a wide-area network. Previous protocols are either slowdown tolerant with higher latency than leader-based protocols [55, 60], or do not tolerate certain types of slowdowns (§5.1.2) [56].

Guided by observations about proactive and reactive slowdown tolerance (§5.1), AVICENNA is a leader-based and reactive protocol. AVICENNA has a single leader replica that sends Accept messages to replicas that reply AcceptOk. After receiving a simple majority quorum of $f+1$ AcceptOk replies the leader can commit. Thus, when there

are no slowdowns AVICENNA can commit client commands within the same number of message delays as other leader-based protocols like Multi-Paxos and Raft [35, 59]. Previous slowdown tolerant work could commit in more than two message delays even when there was no slowdown, required a delay before sending a message analagous to an Accept that could be up to two message delays long [55], or would incorrectly react to certain types of slowdowns (§5.1.2).

AVICENNA's reaction mechanism is inspired by a protocol from previous work that we will refer to as MR99 [52]; after detecting the slowdown it can change leaders in one wide-area message delay from the closest quorum to the next leader, which is deterministic. AVICENNA uses the idea of a *delegate*, giving more voting power to change leaders to a closer replica that can react faster, to react faster than a quorum would be able to. The intuition is to transfer leader-change votes from replicas far from the leader to a replica close to the leader. For safety the far replicas cannot vote to Accept commands, but because they are far from the leader they are unlikely to be in the leader's Accept quorum. The close replica, the delegate, can use the far replicas' leader-change votes to be able to change leaders in one message delay from the delegate to the next leader (or from any replica to the next leader if the next leader is the delegate).

Guided by previous work, Latent Copilot [56], that could incorrectly react to certain types of slowdowns, instead of actively detecting slowdowns which is difficult (§5.1.2), AVICENNA asks the counterfactual question "What would the latency be if another replica was the leader?" To answer this question AVICENNA uses a *ghost protocol* with a ghost leader that executes the same protocol as the real protocol except for execution of commands. The ghost protocol allows AVICENNA to explicitly answer the counterfactual question; naturally tolerating slowdowns that are difficult to detect actively and accurately.

AVICENNA enables applications to configure the *objective function*, which determines when the ghost leader is better. AVICENNA passes real and ghost latencies to the objective function which determines whether or not AVICENNA should change leaders to the ghost

leader. For example, the objection function could be to minimize the median or maximum request latency.

To reduce the overhead of running multiple protocols in parallel on the same set of replicas AVICENNA supports configurable counterfactual evaluation: the application may decide which client commands are processed through the ghost protocol. Configuring how often and for which commands are ghost-processed allows the application to explicitly tradeoff how quickly the protocol can react to a slowdown for higher throughput and lower resource utilization.

Through evaluation on 7 geodistributed replicas on Microsoft Azure we expect to find that normal case latency is comparable to Multi-Paxos and lower than Copilot. We further show that AVICENNA is slowdown tolerant in the wide-area under long-lived slowdowns: for example injecting a slowdown of 68 ms results in fast leader-change bounding the change in max client latency.

This dissertation makes the following contributions:

- The contiguous multi-sequence abstraction, which simplifies building correct services compared to the previous noncontiguous multi-sequence abstraction (§2).
- The design of MASON, which provides the contiguous multi-sequence abstraction and is the first multi-sequence design that is scalable (§3).
- A redefinition of slowdown tolerance that is applicable to both the wide-area and local-area and enables comparing slowdown tolerance properties across protocols (§4).
- An examination of proactive and reactive slowdown tolerant protocol designs (§5.1).
- AVICENNA, the first slowdown tolerant protocol that correctly reacts to slowdowns and matches the latency of the current state-of-practice in the wide area (§5).

Chapter 2

The Contiguous Multi-Sequence

This section is an orientation to the multi-sequence abstraction. Section 2.1 explains how to build strongly-consistent services with the generic multi-sequence abstraction. Section 2.2 describes why building services with the existing *noncontiguous* multi-sequence abstraction is challenging. Our *contiguous* multi-sequence abstraction instead makes it easy to use multi-sequences to build scalable, consistent services.

2.1 Building Services with Multi-Sequences

The sequence abstraction globally orders operations in a single sequence space. The multi-sequence abstraction extends the sequence abstraction to multiple sequence spaces to enable the service to order operations only when they execute on the same subset of data. This enables services to execute operations in order with less coordination: servers managing a subset of data only need local ordering information, reducing contention compared to ordering all operations globally, improving throughput and latency. Services built on the generic multi-sequence abstraction typically include clients, a sequencing component, and servers, each holding one or more shards. Typically, each shard stores a subset of the service's data and is replicated for fault tolerance. Each shard has its own sequence space, a sequence of strictly increasing integers that order operations on the shard's data. To execute

an operation, a client identifies the shards involved in the operation, gets a multi-sequence number from the sequencing component with one number from each relevant shard's sequence space, and sends the operation to the shards' servers with the multi-sequence number. Each server locally uses the multi-sequence number to order this operation's data accesses relative to other operations' accesses. In contrast, with a single, global sequence space, each shard would need to know the next operation to execute across all shards instead of only the next operation concerning its own data.

We next define multi-sequence numbers, explain how they are assigned to operations consistently, and describe how execution protocols use them to scale execution.

Multi-sequence numbers. A *multi-sequence number*, n, is a set of $\langle ssid, sn \rangle$ tuples where *ssid* is a unique number identifying the sequence space, and *sn* is a sequence number in that space. The sequence number in space s in multi-sequence number n is denoted n_s. For a set of sequence spaces requested by a client, the sequencing component returns a multi-sequence number consisting of the next sequence number n_s in each relevant space s.

Strictly serializable multi-sequence number assignment. From clients' perspectives, strictly serializable services process operations one at a time in an order that a single machine could have received them [61]. Concretely, *strict serializability* requires that there exists a legal total order of operations consistent with the partial ordering of "real-time" precedence, i.e., if a completes before b begins, then a must be ordered before b [25, 61].

Multi-sequence numbers enable strongly consistent distributed services when assigned to operations in a strictly serializable order. To simplify discussion, we define a default, Δ, where $n_s = \Delta$ for all n_s not mapped to a specific sequence number (i.e., all s not in this multi-sequence number). For the set of all sequence spaces S, we define a partial ordering over all multi-sequence numbers where $a < b \iff \forall s \in S, a_s \neq \Delta \wedge b_s \neq \Delta \implies a_s < b_s$. The multi-sequence abstraction guarantees that two multi-sequence numbers either share no common sequence spaces or are strictly ordered (i.e., if $a_s < b_s$ for one common space

s, then $a_{s'} < b_{s'}$ for all common spaces s', implying $a < b$). The partial ordering of the multi-sequence numbers defines the ordering of operations. If strict serializability imposes an ordering between two operations, then multi-sequence numbers assigned on path with their execution capture that ordering.

Execution protocols. To use the multi-sequence abstraction, a service developer implements an *execution protocol* that executes operations in order of their multi-sequence numbers, yielding a strictly serializable service. The execution protocol runs on clients (typically encapsulated in a client library) and on the service's servers. For clients, the execution protocol defines how operations are mapped to the service's shards and which sequence spaces are involved in a given operation. For servers, it determines when shards can safely execute operations, based on the operations' multi-sequence numbers.

Scalable execution. Multi-sequence numbers enable services to scale throughput up to the rate the sequencer can assign sequence numbers. Execution scales through parallelism: when some shards are executing an operation, other shards can execute a different operation. The sequence spaces in multi-sequence numbers determine which operations can execute in parallel, as operations with disjoint multi-sequence numbers access different shards. As long as multi-sequence number assignment keeps up, the service can increase its throughput by adding more machines and creating more shards. However, existing multi-sequenced services use *monolithic* (single-machine) sequencers, which can never assign sequence numbers to operations at a higher rate than a single machine can support and hence limit the service's scalability.

2.2 From Noncontiguous to Contiguous

The generic multi-sequence abstraction is realized as a *noncontiguous* abstraction in existing services, which use it to expose higher-level abstractions [44, 71]. As we explain

next, noncontiguity complicates service development. In contrast, the *contiguous* multi-sequence abstraction simplifies developing services with multi-sequences by encapsulating that complexity within the abstraction.

Holes in a noncontiguous sequence complicate the abstraction. *Holes* occur when a sequence number is not used for an operation. For example, a hole occurs if a client fails after receiving a sequence number but before using it. A shard may see, e.g., sequence numbers 1–3 and then receive an operation with sequence number 5, indicating a potential hole at 4. To preserve strict serializability, the shard may only execute operation 5 after 4 is used, since 4 could belong to any operation. To make progress in the absence of an operation, the service must decide that the entire multi-sequence number is a hole and enforce that it is not used on any shard, typically by assigning a *no-op* to each of its sequence numbers.

Handling holes complicates service design. The service must have a mechanism to identify sequence numbers that are potential holes. Existing designs use timeouts [5, 71] or infer holes from out-of-order operation arrival [44, 71]. More challenging is that the service's servers must reach service-wide consensus on whether a sequence number is a hole, then coordinate to ensure that the other numbers in the hole's multi-sequence number are treated as holes to avoid partially executing a cross-shard operation. Existing services achieve this with a global shared log [71] or a failure coordinator [44]. Requiring consensus in the execution protocol makes a service developer's task significantly more difficult. Consensus is hard to implement and incorporate [9, 60], and requires developers to understand the nuances of the sequencing component and consensus implementation in depth.

Although existing services feature workable solutions for handling holes, requiring services to select and properly incorporate a solution does not reflect operational best practices. Much of the purpose of providing infrastructure building blocks (such as an implementation of the multi-sequence abstraction) is to enable services to use them without needing to understand their complexities, via clean abstractions that mask the subtleties

of their internal operation and failure modes. Pushing the complexity of handling holes to services increases the chances of one doing so incorrectly, similar to how pushing memory management to individual programmers increases the chances of memory leaks.

Our contiguous multi-sequence avoids holes and hides consensus. Our abstraction assigns exactly one operation to each sequence number in each sequence space. Service developers can focus on designing execution protocols that achieve their services' goals, a much simpler task when freed from reasoning about holes or implementing consensus. Eris [44] and vCorfu [71], the two existing designs built on the noncontiguous multi-sequence abstraction, were developed by distributed systems experts. With the contiguous multi-sequence abstraction, we aim to empower developers without such expertise to use multi-sequences to build scalable, consistent services, and make it easier and faster for experts.

Chapter 3

MASON

3.1 MASON Overview

The central contributions of MASON are to shield services from the complexity of dealing with holes by providing the contiguous multi-sequence, and to provide the benefits of the multi-sequence abstraction while allowing ordering throughput to scale beyond what a monolithic sequencer can provide. Section 3.2 describes how the components work together to guarantee a contiguous multi-sequence. Section 3.3 describes how MASON enables scalability with two mechanisms that relieve all throughput bottlenecks.

3.1.1 Model and Assumptions

We assume a set of processes that communicate via point-to-point communication over an asynchronous network, where messages can be arbitrarily delayed and reordered. We assume a crash failure model, where processes execute according to their specification until they cease sending messages and the failure is undetectable to other processes. MASON is safe under these assumptions. We assume service shards implement at-most-once semantics to handle retransmissions.

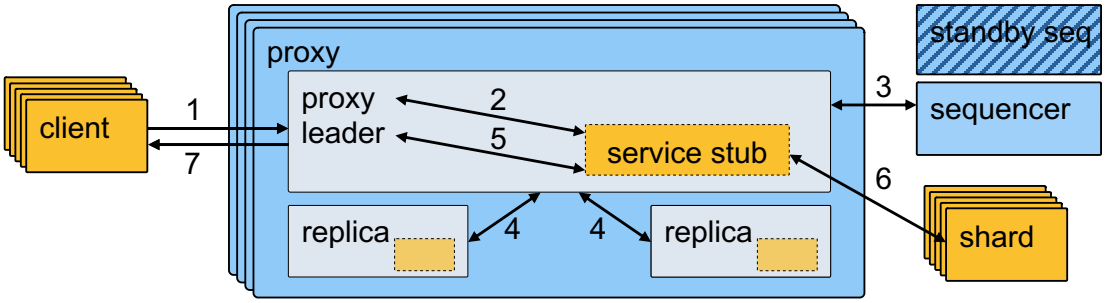


Figure 3.1: The components of a service built with MASON and an operation's flow through the service. Blue components are part of MASON; yellow components are supplied by the service. Numbers correspond to steps in §3.1.3.

3.1.2 MASON Components

Figure 3.1 shows how MASON is used in a service. It also shows MASON's two types of internal components: a *sequencer* and replicated *proxies*. The core of MASON's design is a monolithic sequencer that provides high-throughput operation ordering, surrounded by a replicated proxy layer that handles the failure modes and bottlenecks impeding existing sequencers.

The sequencer allocates increasing multi-sequence numbers. It is implemented by a single machine, and only one sequencer is active at a time. MASON maintains a cold *standby sequencer* for failure recovery.The standby sequencer does not participate in normal case operation and is only required for liveness. The monolithic sequencer at MASON's core provides the benefits of existing sequencers: contention-free, high-throughput ordering of operations in a distributed system. In our system, MASON itself is the distributed system, leveraging the sequencer's benefits while managing its drawbacks to provide a simpler, scalable building block to the service.

Proxies are replicated state machines (RSMs). We treat the RSM protocol as a black box, but for liveness require that it is leader-based and that the protocol informs a replica when it gains and loses leadership; these two properties enable a leader takeover protocol that allows MASON to only replicate once, discussed below (§3.2.1). Both requirements could be eschewed at the cost of normal-case latency by replicating state both before making

a request to the sequencer and after. However, because leader-based protocols are common in practice, we choose the lower latency design. Our implementation uses Raft [59]. A proxy is thus logically a single entity implemented by a leader process and multiple follower processes on separate machines. The leader accepts operations from clients and executes them via the *service stub* using multi-sequence numbers. The rest of this paper refers to a proxy replica group simply as a *proxy*. A MASON deployment may have one or more proxies, depending on system load.

Identical to many other RSM-based systems, we assume at most f of $2f+1$ proxy replicas fail [35, 45, 51, 59]. A MASON deployment must be configured so that f is sufficiently large. In the rare event that more than f machines fail, manual intervention by an operator is necessary to restore availability.

Service stubs are implemented by the service built on MASON and drive the execution protocol on the proxies. Service developers interact with MASON on the proxies through service stubs which execute within the proxy's process. When a proxy receives an operation from a service's client, it passes the operation to the stub. The stub either requests that MASON order the operation, or executes the operation immediately if it need not be ordered, e.g., an inconsistent read. After ordering and replicating the operation, the proxy returns it back to the stub which begins the execution protocol. Stubs are analogous to client libraries in existing multi-sequenced services. Section 3.4 shows how stubs are used to develop services.

The proxy may *batch* requests for multi-sequence numbers for scalability, i.e., request multi-sequence numbers for multiple client operations in one sequencer request (§3.3). The sequencer *allocates* a multi-sequence number for each operation in the batch. An allocated multi-sequence number is one given to a proxy that the sequencer promises not to allocate again. Proxies are responsible for *assigning* multi-sequence numbers to client operations. Assignment uses replication to permanently associate a multi-sequence number with an operation and guarantee it will never be assigned to another operation. Once the proxy

has replicated the assignment of a multi-sequence number to an operation, it returns the operation and multi-sequence number to the service stub for execution.

3.1.3 Normal-Case Operation of MASON

The normal case operation of MASON, shown in Figure 3.1, includes the following steps:

1. A client sends an operation to a proxy.
2. The proxy passes the operation to the service stub which determines the relevant sequence spaces.
3. The proxy asks the sequencer to allocate a multi-sequence number covering the relevant sequence spaces.
4. The proxy replicates the allocated number and operation, assigning the number to the operation.
5. The proxy returns the operation and multi-sequence number to the service stub.
6. The service stub and shards run the execution protocol.
7. The proxy sends the response from the stub to the client.

3.2 Ensuring a Contiguous Multi-Sequence

MASON provides a contiguous multi-sequence by handling all potential sources of holes: client failures (Figure 3.2), network drops (Figure 3.3), sequencer failures (Figure 3.4), and combinations thereof. This section covers how MASON handles each of these failure scenarios and then sketches a proof of strict serializability.

3.2.1 Proxies Prevent Holes from Client Failure

In a multi-sequenced service, client failure can cause holes when the client obtains a sequence number and fails before using it in the service. For instance, Figure 3.2 illustrates Client A failing before using sequence number 3 from sequence space *i*, resulting in a hole at 3. MASON prevents such holes with proxies that manage multi-sequence numbers on clients' behalf. Proxies are replicated for fault tolerance, eliminating this source of holes. A proxy will always return an operation that was assigned a multi-sequence number to the service stub even if the client fails and even if a minority of proxy replicas fails.

A byproduct of replication is that proxies maintain a record of every assigned sequence number, which is used in sequencer recovery (§3.2.3). By masking client failure and maintaining state needed for sequencer recovery, proxy replication is a key mechanism for avoiding holes in MASON.

The proxy replication strategy is driven by correctness and performance. Proxies must replicate enough information to preserve contiguity and strict serializability. Replicating every input to the proxy leader would be correct, but this would add unacceptable latency to client requests and burden proxies with excessive communication overhead. Fortunately, MASON can skip replication for all but one step in operation processing, because the other steps can be safely retried, including after client, sequencer, and/or proxy replica failure.

The exception is step 5 (Fig. 3.1), returning a multi-sequenced client operation to the service stub. Replicating the mapping of each client operation to a multi-sequence number

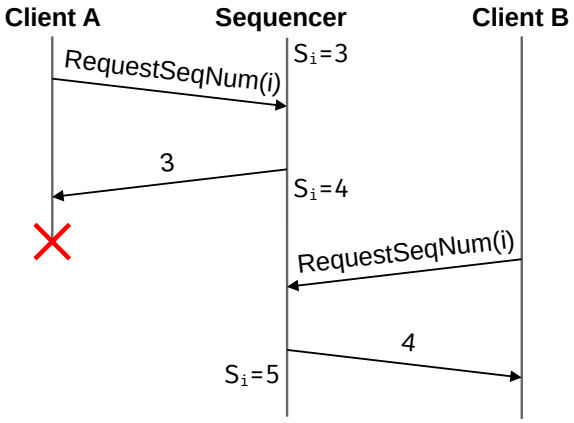


Figure 3.2: Hole caused by client failure.

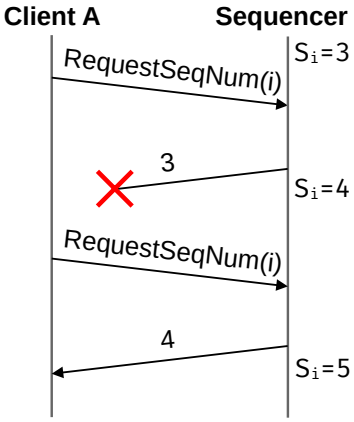


Figure 3.3: Hole caused by network drop.

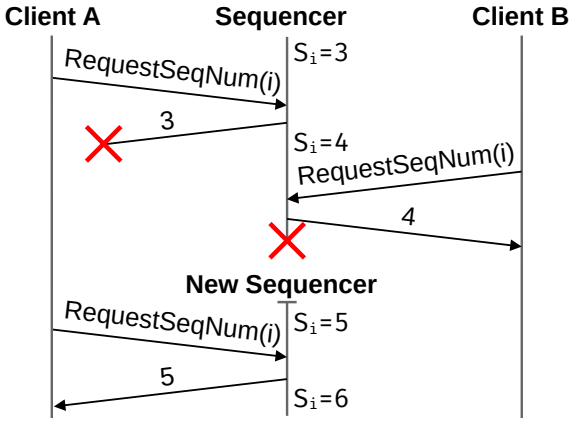


Figure 3.4: Hole caused by sequencer failure.

Figure 3.5: Potential sources of holes. S_i is the next sequence number in sequence space i.

before this step is critical for correctness in MASON. Suppose the mapping is not replicated. The sequencer and proxy leader could fail concurrently after the leader returns a multi-sequenced client operation to its service stub, but before the stub sends its operation to every relevant shard. The shards that received the operation may execute it, but the operation will not be completed after recovery because the mapping of multi-sequence number to operation was lost. Exposing the partial execution violates strict serializability. Therefore, before returning an operation to the service stub, the proxy must permanently associate the operation with a multi-sequence number through replication. Once replication succeeds, the sequence number is *assigned* to the operation.

We next describe how the proxy processes operations, in order to explain why all other steps are safe to retry. We discuss one operation and a single sequence space for ease of explanation; the reasoning can be easily extended to batches of operations and multiple sequence spaces.

Receiving a client operation. Clients can send an operation to any proxy. When a proxy leader receives an operation from a client, it passes the operation to the service stub. If the stub requests that the operation be ordered, the leader allocates a *sequencer request ID* for that operation (step 3 in Figure 3.1). Sequencer request IDs are allocated only by the leader, so they are trivially contiguous and strictly increasing. Sequencer request IDs are used during proxy failover to recover sequence numbers that were allocated but not yet assigned to any operation, i.e., potential holes.

Requesting a sequence number. The leader then requests a sequence number from the sequencer, with the sequencer request ID (step 3). If the sequencer has not seen this sequencer request ID from this proxy, the sequencer updates its state in two relevant ways: it allocates a sequence number for this request by incrementing the sequence counter in the requested sequence space, and maps the sequencer request ID to the allocated number.

If the sequencer has seen the sequencer request ID before, it responds with the previously allocated sequence number and marks it as a retransmit.

Proxy leader failure. When a proxy leader fails, the new leader must recover the sequence numbers that were allocated but not yet assigned. The state needed to correctly match allocated but unassigned sequence numbers to operations was lost with the failed leader, so these are temporary holes. We now explain how we use sequencer request IDs to recover such holes. This is the key mechanism for ensuring correctness when proxies execute only one round of replication.

The new leader collaborates with the sequencer to identify these temporary holes as follows:

1. The new leader saw a contiguous set of sequencer request IDs until some ID x, after which it saw noncontiguous IDs until y. The range from x to y is noncontiguous because the leader replicates sequence number-operation pairs as they arrive from the sequencer, which may be out of order.

2. The new leader requests sequence numbers for all IDs from $x+1$ until $y-1$ that were not replicated. The sequencer will either return already-allocated sequence numbers, or will allocate new numbers for the IDs.

3. The new leader replicates and assigns all returned sequence numbers to no-ops and returns them to the service stub.

4. The new leader then resumes normal operation, allocating sequencer request IDs from $y+1$.

There may be allocated but unassigned sequence numbers with sequence request IDs greater than y. In such cases, the sequencer will mark the returned sequence numbers as retransmits. The leader replicates and assigns them to no-ops and retries the request with a new sequencer request ID. If the sequencer fails concurrently with leader failure, the

sequencer recovery protocol recovers and assigns no-ops to any allocated but unassigned sequence numbers (§3.2.3).

Returning the operation and sequence number to the service stub (step 5). Strict serializability dictates that the service's execution protocol cannot use one sequence number for multiple operations, and different sequence numbers cannot be used for one operation. MASON must therefore guarantee the sequence number associated with an operation never changes once the service is made aware of it. MASON thus replicates the sequence number-to-operation assignment (step 4) before passing the operation to the service stub.

The proxy leader's other steps in handling a client operation—passing the operation to the service stub and forwarding the service's response to the client (step 7)—can be safely left unreplicated. Retrying these steps is safe. The service stub, shards, and clients already provide at-most-once semantics to handle retransmission due to network drops, so they will be able to handle retransmission from the proxies.

3.2.2 Reliable Transport Prevents Holes from Packet Loss

Holes can occur after network drops (as illustrated in Figure 3.3). MASON handles network drops with a reliable transport layer. Since the state needed to reliably transport multi-sequence numbers is lost on sequencer failure, MASON uses a recovery protocol to correctly fill holes with no-ops in case of simultaneous packet loss and sequencer failure (§3.2.3). Reliable transport and the sequencer recovery protocol ensure that every allocated multi-sequence number arrives at a proxy.

3.2.3 Recovering to Prevent Holes from Sequencer Failure

Sequencer failure can cause holes if failure occurs before the reliable transport protocol can retransmit a dropped response. For instance, suppose the sequencer allocates and sends the sequence number 3 for sequence space *i* and later 4 for the same sequence space *i* for

two client operations, as illustrated in Figure 3.4. If the message containing 3 is dropped and the sequencer fails before retransmission, but a client receives 4, then 3 is a temporary hole. One solution replicates the sequencer to permanently associate client requests and multi-sequence numbers. However, replication compromises the main benefit of a sequencer: simplified ordering so the sequencer can devote all its resources to allocating numbers.

MASON instead runs one active sequencer, backed by an idle standby sequencer and sequencer recovery protocol. If the active sequencer fails, the standby sequencer takes over and executes the recovery protocol to correctly fill any temporary holes caused by the failure, ensuring a contiguous multi-sequence when the standby resumes normal operation.

MASON's sequencer recovery protocol is based on two observations. First, the proxies' collective state includes which sequence numbers have been assigned, so they collectively know where potential holes in each sequence space are. MASON assigns these sequence numbers to no-ops. Second, all outstanding operations are concurrent. An outstanding operation is one that a proxy received (step 1 in Fig. 3.1), but has not yet assigned a sequence number (step 4), and thus is not ordered. When the standby sequencer resumes normal operation, it can allocate new multi-sequence numbers for outstanding operations in any relative order, as long as they are ordered after the highest previously-assigned sequence number in each sequence space, which the proxies collectively know.

The steps in MASON's sequencer recovery protocol are:

a) Detect sequencer failure and activate a standby sequencer.

b) Identify potential holes in each sequence space.

c) Replicate the assignment of no-ops to holes.

d) Resume normal operation with new sequence numbers.

Failure detection and standby sequencer activation. Proxies unreliably detect sequencer failure with timeouts and pings. If a proxy does not hear from the sequencer after a timeout

(.5 s in our implementation), it pings the sequencer. After another timeout, the proxy declares the sequencer failed and initiates recovery by activating the standby sequencer. The standby sequencer informs the other proxies that recovery has begun. All proxies then replicate a special recovery operation and seal their sequence spaces, rejecting any packets from the previous sequencer. The new sequencer waits for all proxies to complete the sealing process before resuming recovery. Replicating the recovery operation on all proxies before allowing the standby sequencer to resume recovery ensures proxies reject all packets from the previous sequencer. This in turn, ensures there is only one active sequencer at a time even when proxy leaders fail, sequencer-failure detection is incorrect, or messages from the previous sequencer were delayed or reordered in the network.

Identifying potential holes. During normal operation, proxies track their *local views* of each sequence space. A proxy's local view is the subsequence of numbers in each sequence space that the proxy has assigned to operations. After sealing, proxies send their local views to the standby sequencer. The standby sequencer reconstructs each sequence space, exposing any temporary holes. Garbage collection of proxies' local views is described at the end of this section.

Assigning temporary holes to no-ops. The standby sequencer notifies proxies of any temporary holes in each sequence space. Proxies assign these sequence numbers to no-ops, replicate the assignment, and pass them to the service stubs, as they would with client-issued operations.

Resuming normal operation. The standby sequencer identifies the start of each sequence space based on the highest number in each sequence space compiled from the proxies. It then notifies proxies to resume normal operation and allocates new sequence numbers from that point. Proxies must re-request sequence numbers for all outstanding operations.

Concurrent proxy leader failure. If a proxy leader fails during sequencer recovery, the new leader will have enough state to correctly participate in recovery. In particular, if the leader fails before replicating the recovery operation, the sequencer will not have started recovery, and all holes normally recovered as part of leader recovery will be found when the sequencer is rebuilding the sequence. If the leader fails after replicating the recovery operation, the new leader will have the same state as the previous leader when the previous leader began sequencer recovery.

Garbage-collecting sequence number tracking state. Proxies run a lightweight garbage collection protocol to discard tracked sequence numbers that are no longer needed for sequencer recovery. Each sequence space is partitioned into intervals of size N. When all N sequence numbers in an interval have been assigned to operations, it is safe to discard the state associated with those sequence numbers. To determine when all N numbers have been assigned, the proxies form a communication ring and periodically send an accumulating count of the sequence numbers assigned in each sequence space's latest interval. At the end of a round, if any sequence space's count is N, the interval is completely assigned; all state associated with that interval is discarded.

3.2.4 Proof Sketch of Strict Serializability

This subsection sketches a proof of the strict serializability of the assignment of multi-sequence numbers to operations. We include the formal proof in the appendix (§A). We make the assumptions stated in §3.1.1. Our proof reasons about pairs of operations, showing they are either *strictly concurrent*, where they do not share sequence spaces, or *strictly ordered*, where if $a_n < b_n$ for some overlapping sequence space n, then $a_{n'} < b_{n'}$ for all overlapping sequence spaces n' , where a_n denotes the sequence number in sequence space n assigned to operation a.

To show that there exists a total order over all completed operations consistent with the partial ordering of real-time precedence, we exhaustively analyzed all cases of failure scenarios from no failures to concurrent failure of proxy leaders, proxy followers, and sequencer. In all cases an operation is assigned at most one multi-sequence number which occurs if/when replication to a majority of replicas in a proxy succeeds. The assigned multi-sequence numbers for all operations that access overlapping sequence spaces are then strictly ordered by either the same sequencer, or by an initial sequencer and a standby sequencer that recovers all previous assignments before allocating any new multi-sequence numbers. Thus, the partial order of assigned multi-sequence numbers strictly orders all conflicting operations. Further, this partial order is consistent with real-time precedence either trivially when two operations are ordered by the same sequencer or because a standby sequencer only allocates numbers larger than the maximum previously assigned in each sequence space. Only strictly concurrent (i.e., no overlapping sequence spaces) operations are unordered by that partial order, and any ordering of them results in a valid total order. Extending the partial order to a total order consistent with real-time precedence is thus trivial: unordered operations are first ordered by the partial order of real-time precedence and then remaining unordered operations are arbitrarily ordered.

3.3 Supporting Scalable Throughput

A service's achievable throughput (*service throughput*) is capped by the minimum of the rate at which it can execute requests (*execution throughput*) and the rate at which it can order requests (*ordering throughput*). Execution throughput scales when more service shards are added if and only if the service implements a scalable execution protocol. Ordering throughput scales only if the ordering component scales. Previous multi-sequence abstraction designs do not scale.

MASON supports scalable service throughput by removing the bottlenecks that limit monolithic-sequencer designs and achieving scalable ordering. This section describes two complementary mechanisms that alleviate all ordering throughput bottlenecks: horizontally scaling out the proxy layer, and batching requests to the sequencer.

Potential ordering throughput bottlenecks. MASON has two components, so there are two potential bottlenecks on computation: the proxy layer and the sequencer. Each component sends and receives network traffic, so there are four potential bottlenecks on network bandwidth. Our two scaling mechanisms address all six bottlenecks: scaling out the proxy layer relieves all bottlenecks at the proxy layer, and batching relieves all bottlenecks at the sequencer.

The proxy layer scales out. When MASON is bottlenecked by a proxy layer resource, the proxy layer can scale out. Each proxy operates essentially independently, so holding all else constant, doubling the number of proxies doubles the amount of computation and bandwidth available at the proxies for processing client operations, doubling the proxy layer's achievable throughput.

In truth, proxies are not completely independent; there is overhead to garbage collect multi-sequence number tracking state (§3.2.3). However, the overhead is constant for each

proxy with respect to the number of proxies due to the ring communication pattern; thus, it does not affect the proxy layer's scalability.

In §3.1–3.2 we have assumed a static configuration where the numbers of proxies and shards do not change. MASON components can be reconfigured as follows. To add a new proxy, the new proxy first creates a connection to the sequencer and then joins the garbage collection ring using standard techniques, e.g., those used in distributed hash tables [66]. Removing proxies is more difficult to do safely. For example, if a proxy is removed and the sequencer fails, the recovery protocol may not be able to reconstruct a complete view of the used sequence numbers (i.e., it will be missing those numbers used by the removed proxy but which were not yet garbage-collected). It may attempt to assign those used sequence numbers to no-ops, which is not safe. Thus, to remove a proxy, the proxy stops processing client requests, but continues to take part in garbage collection until all sequence numbers the proxy received are garbage collected. At this point the proxy can remove itself from the ring and disconnect from the sequencer. Waiting until all of its numbers are garbage collected ensures any used multi-sequence number will not be assigned a no-op. Alternatively, the proxy could transfer all of its sequence numbers to a different proxy, e.g., the next proxy in the garbage collection ring, and then leave the ring. Reconfiguring the service's shards can be achieved through operations internal to the service and via the service stubs.

Batches are as efficient as single requests. When MASON is bottlenecked by the sequencer proxies can increase throughput by batching multi-sequence number requests. This batching is perfect, holding all else constant, in that a request for one client operation uses the same resources as a request for multiple operations.

To request multi-sequence numbers for a batch of client requests, the proxy constructs a sequencer request which indicates the relevant sequence spaces and how many numbers are required from each sequence space to order the operations in the batch and sends a single sequencer request for the batch. The sequencer allocates the requested count of

sequence numbers in each sequence space and replies with the lowest allocated number in each sequence space. Finally, the proxy iterates through client operations in the order they were received and gives each operation the next lowest sequence number in each of its sequence spaces.

MASON alleviates all bottlenecks on the sequencer by increasing the batch size. MASON's batching is timeout-driven: all client requests that arrive at a proxy within the timeout are batched together. By doubling the timeout (hence batch size) at a given client load, proxies can halve the rate at which they issue sequencer requests. The sequencer, in turn, would need half the resources to handle the same client load. The sequencer can thus handle twice the ordering throughput before hitting the same bottleneck. Timeout-driven batching is naturally dynamic: higher client load results in larger batches.

Why not batch at clients? A strawman design for increasing ordering throughput is to batch requests at clients, which has two limitations. First, the maximum throughput is limited by the number of parallel requests a client will individually make. Second, batching at clients requires waiting until the client has issued those requests, which can substantially increase latency. In contrast, MASON's proxies can batch across any number of clients, achieving the large batches that allow it to scale. In general, naïvely adding only a batching layer to prior designs does not work, as it introduces new failure modes (e.g., batching machine failure) that require a comprehensive service redesign such as that of MASON.

3.4 Services

This section explains how services can easily use MASON and its contiguous multi-sequence abstraction to scale service throughput. We describe two services we implemented over MASON: a distributed shared log based on CORFU [5] and a distributed prototype of the coordination service ZooKeeper [26].

3.4.1 Interaction with MASON

A service's execution protocol consists of (at least) two components: *shards* and *service stubs*. Shards are implemented entirely by the service and interact with service stubs and other service-implemented components. Service stubs are the mechanism by which services interact with proxies. They determine an operation's relevant sequence spaces and request ordering via MASON if necessary, drive the execution protocol interacting with other service components, and have control of the operation until informing MASON that the operation is complete. This is sufficient for the services we implement here; more complex services may need multi-round sequencing for some operations, e.g., where the write set depends on the read set. In that case, MASON could be augmented so that the stub could request another round of ordering and include metadata, which MASON replicates and the service can use to resume execution if the current proxy leader fails.

3.4.2 Making CORFU Scalable: Corfu-MASON

CORFU is a shared log supporting append and read operations that consistently execute across shards [5]. Appends write a value to the current tail of the log. Reads return the value written to a specified log position. Many applications can be implemented with shared logs, e.g., producer-consumer queues and logging [28, 64].

We use MASON to implement Corfu-MASON, a service based on CORFU. CORFU's original implementation does not scale; although CORFU has a scalable execution protocol,

the implementation is limited by the ordering throughput of its monolithic sequencer [5, 71]. By replacing the sequencer with MASON, MASON's scalable ordering combines with the scalable execution protocol to enable the whole service to scale.

Corfu-MASON uses CORFU's scalable execution protocol. The shared log is represented by a single sequence space. Appends acquire a sequence number that directly determines which log position to write. A round-robin mapping of log position-to-shard ensures append load is uniform on shards, enabling appends to execute in parallel [5].

Corfu-MASON implements two of CORFU's three operations, `append(`b`)` and `read(`l`)`. `append(`b`)` appends the entry b to the log and returns the log position l to which it was written. `read(`l`)` returns the entry at log position l, or an error code if the entry does not exist. CORFU implements a third operation, `fill(`l`)`, to fill holes in the sequence (and the log) caused by failed clients. CORFU clients detect holes in the log with a timeout and execute `fill(`l`)` to fill the lth position with junk. The timeout-and-`fill(`l`)` procedure is unnecessary in Corfu-MASON because of MASON's contiguous sequence.

Corfu-MASON's execution protocol uses sequence numbers for `appends` to determine which log positions to write, which in turn map to specific shards. In addition to eliminating the need for `fill` operations, MASON's contiguous sequence simplifies `reads`. If a client attempts to read a log position that has not been written yet, it can simply keep checking that log position. The contiguous sequence guarantees that the entry will eventually be written. `reads` need not be ordered and hence are not ordered or replicated by MASON; the service stub executes reads immediately. CORFU tolerates shard failure using client-driven chain replication [70], and so Corfu-MASON uses service stub-driven chain replication.

Corfu-MASON was implemented in a single day thanks to both the simplicity of CORFU and the strong abstraction of a contiguous sequence provided by MASON.

3.4.3 Making ZooKeeper Scalable: ZK-MASON

ZK-MASON is a ZooKeeper-like coordination service built on MASON. ZooKeeper [26] is a widely-used coordination service implemented on ZooKeeper Atomic Broadcast (ZAB) [29], a version of state machine replication (SMR). ZAB, like other SMR protocols, cannot scale: it is fundamentally limited by the rate a single machine can execute requests. Furthermore, ZooKeeper uses a single replicated state machine to ensure consistency, so an instance cannot be sharded. We designed ZK-MASON to be scalable, using the cross-shard consistency and scalable ordering provided by MASON.

ZK-MASON operations. Similar to ZooKeeper, ZK-MASON maintains a set of *znodes*. Each znode has a pathname beginning with "/" (similar to a filesystem) and data associated with it. We implemented seven operations in ZK-MASON:

- `create(`*path*`,`*data*`,`*flags*`)`: creates a znode with pathname *path* and data *data*. *flags* allows the client to specify a persistent or ephemeral znode.
- `setData(`*path*`,`*data*`,`*version*`)`: sets the data at *path* if *version* matches the current version, or if *version* is -1.
- `getData(`*path*`,`*watch*`)`: gets the data at *path*.
- `exists(`*path*`,`*watch*`)`: checks if the znode exists.
- `delete(`*path*`,`*version*`)`: deletes znode specified by *path* if *version* matches the current version, or if *version* is -1.
- `getChildren(`*path*`,`*watch*`)`: returns the children of *path*

The read operations `getData`, `exists`, and `getChildren` return the znode's current version. Read operations have a *watch* flag, which sets a *watch* on the znode if the flag is set. ZK-MASON watches have the same semantics as ZooKeeper watches. Watches are triggered

by updates depending on the type of read operation and the type of update operation. For example, a watch set by `getChildren` is triggered after a `create` or `delete` of a child, but not by any `setData` on its children, as that does not change the result of `getChildren`. ZK-MASON notifies the client when its *watch* is triggered.

ZK-MASON execution protocol. ZK-MASON's execution protocol is based on Eris's execution protocol [44]. ZK-MASON assigns znodes to shards based on a hash of the full pathname. Shards consist of $2f+1$ servers; each shard tolerates f failures. Each server executes incoming operations in order of the shard's sequence space. When a proxy receives a client operation, the service stub determines which shards are involved in the operation and requests a multi-sequence number for the relevant sequence spaces. For example, to execute a `create`, the service stub hashes the *path* and the parent pathname to get the sequence spaces for those two shards. MASON acquires and replicates a multi-sequence number with the two sequence spaces. The service stub sends a `create` operation to each server in *path*'s shard and an `addChild` operation to each server in *path*'s parent's shard in parallel. When the stub receives a quorum of $f+1$ responses from each shard, the operation is complete; the stub informs MASON of completion, and MASON returns to the client. Read operations, for example, `getData`, only need to receive one response from the shard before returning to the client because they are sequenced and do not change state.

Ephemeral znodes. *Ephemeral znodes* are transient znodes that exist only during an active client connection. They are created by a client and deleted by the service when the client disconnects, either explicitly or due to failure. Ephemeral znodes can be used to add to a distributed queue: if the creating client fails, the object is removed. They can also help manage locks: if a client acquires a lock and fails, the lock is released when the ephemeral object disappears [67]. Implementing ephemeral znodes in ZK-MASON is straightforward. Shards keep a timer that is reset with client heartbeats. After timing out, the shard sends a `delete` to a proxy to delete the node. The `delete` is ordered to prevent divergent shards.

The contiguous multi-sequence abstraction simplifies ZK-MASON. Implementing this service over a noncontiguous multi-sequence would require consensus to deal with holes. Because a missing sequence number could belong to a multi-shard operation, e.g., `create`, the hole-filling consensus would need to be service-wide to avoid partially executing the operation on some shards but not others. To handle cases where aborting a partially-executed operation is impossible, each full operation would need to be persisted by the service so it could be recovered by shards that never received it (e.g., the full operation could be sent to every relevant shard).

In ZK-MASON, if a shard encounters a gap in its sequence space, it can wait for the missing operation and each shard only needs to receive the parts of the operation that will execute on that shard. The contiguous multi-sequence guarantees that the operation will be executed.

3.5 Evaluation

MASON provides two main innovations for building services. First, it is a general, reusable building block that offers the contiguous multi-sequence abstraction. This makes it easy to build efficient implementations of complex services (§3.4). But, as with any such abstraction, we expect overheads compared to specialized implementations. Second, MASON provides a scalable multi-sequence allowing previously unscalable services to now scale. This section quantifies the overhead of MASON's general abstraction for two services (§3.5.2 and §3.5.3), shows MASON provides scalable ordering (§3.5.1), that its scalable ordering does indeed enable services to scale (§3.5.2 and §3.5.3), and that MASON does provide a contiguous multi-sequence despite failures (§3.5.4).

Implementation. MASON is written in C++. All components, including clients, service shards, and MASON components, communicate with eRPC, a reliable RPC framework [31]. eRPC uses unreliable datagrams in Intel DPDK (v. 17.11.5) as its transport layer [17]. We replicate proxies with Raft [59], and periodically durably snapshot their state for Raft log compaction. We do not implement reconfiguration. MASON will be open-sourced by publication time.

Evaluation setup. We evaluate MASON on the Emulab testbed [72] with Dell R430 (d430) machines [14]. We run Ubuntu 18.04.11 with Linux kernel version 4.15.0. The machines have two hyperthreaded 8-core CPUs (Intel E5-2630 "Haswell", 2.4 GHz) with 20 MB L3 cache, 64 GB RAM, and one dual-port 10 GbE PCI-Express NIC (Intel X710).

We load MASON with clients running on separate machines of the same type. Unless otherwise specified, each client machine runs 16 threads, each implementing several logical closed-loop clients that generate new operations as previous operations complete. We control load by varying the number of client machines and the number of logical closed-loop clients per thread. Latency is measured at clients for each operation. We report the median over

five trials of the median latency over all clients in a trial. We present latency as *median/99th percentile*. Throughput is also measured at each client and aggregated over all clients in a trial. For all scalability experiments we derive the throughput by increasing load (i.e., the number of logical clients). We report the highest throughput before latency spikes from overload. We show the median throughput over five trials. Trials are 68 seconds each; the first and last 4 seconds of measurements are discarded.

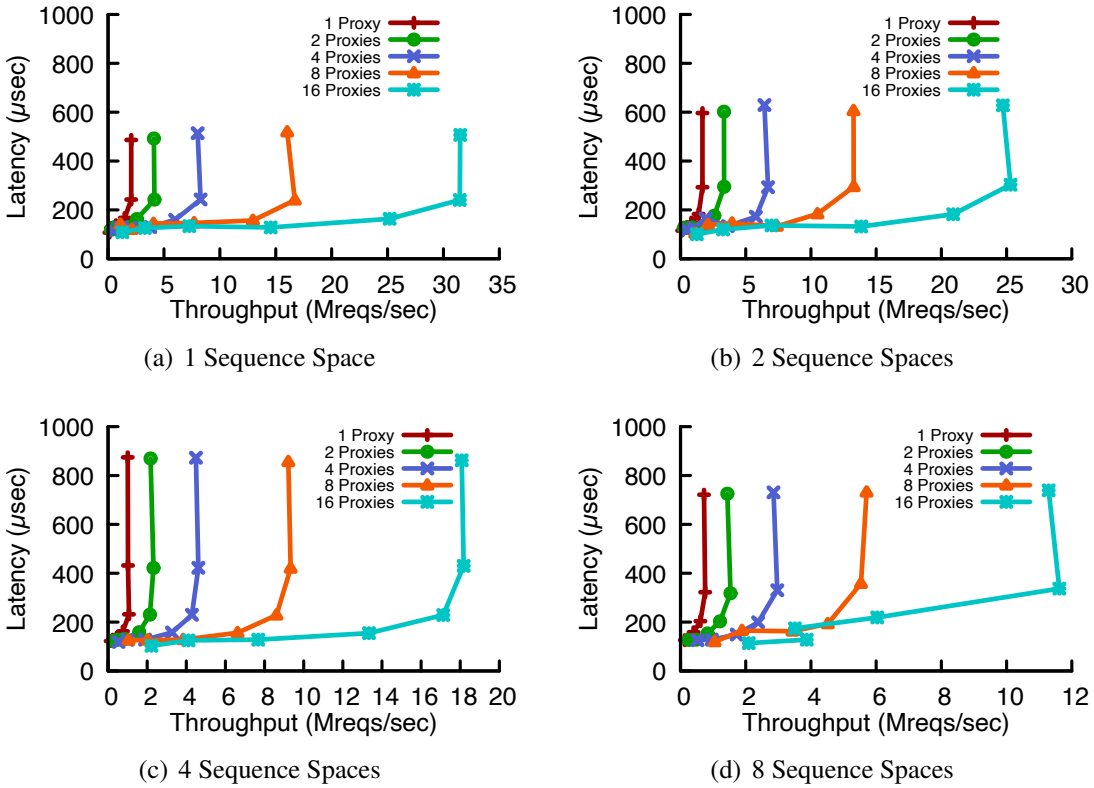
Each proxy is replicated on 3 machines. Experiments in Sections 3.5.1 and 3.5.4 use a stub service with one operation: clients indicate relevant sequence spaces and the service returns the assigned multi-sequence number to the client.

3.5.1 MASON Scales Ordering Throughput

MASON uses two mechanisms to scale ordering throughput: adding more proxies and increasing batching to the sequencer. The first mechanism, adding more proxies, is evaluated in Figure 3.6. Ordering throughput is the number of client operations per second that receive a multi-sequence number and return to clients. To stress ordering throughput, the proxies do not execute operations on behalf of clients in this experiment. We present latency as *median/99th percentile*.

Figure 3.6 shows that, as the number of proxies doubles, the ordering throughput also roughly doubles for each sequence space count. As the number of sequence spaces in the system increases, the per-proxy machine throughput decreases, so overall ordering throughput with the same number of proxies is lower. Latency at these throughputs ranges from ~243 (median)/~380 μs (99th percentile) for a single sequence space to ~358/~693 μs for 8 sequence spaces. This experiment demonstrates that adding more proxies enables MASON to scale ordering throughput.

We are unable to test our second mechanism, increasing batching to the sequencer, because we cannot saturate the sequencer with the machines available on Emulab. With 48 proxy machines, the sequencer processes ~3.2 Mops/s, which is far from the ~14.5 Mops/s



(a) 1 Sequence Space

(b) 2 Sequence Spaces

(c) 4 Sequence Spaces

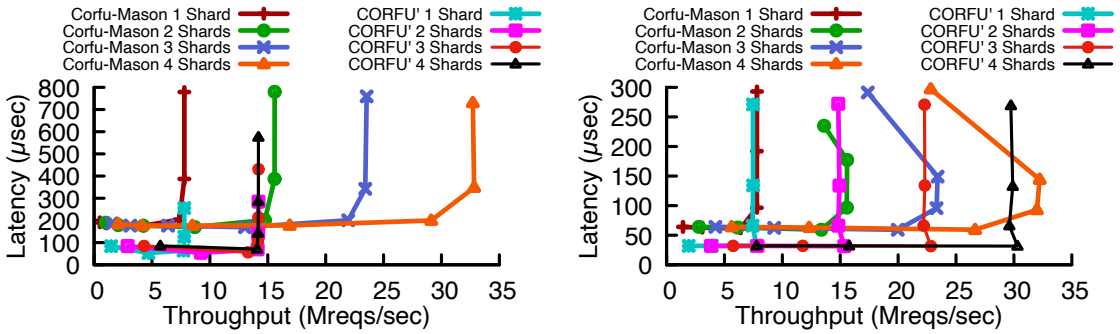
(d) 8 Sequence Spaces

Figure 3.6: MASON ordering throughput-latency; each point represents a given load, doubling the client load from the previous point. MASON scales linearly with the number of proxies: as the number of proxies doubles, the ordering throughput also roughly doubles for each sequence space count.

possible at line rate. As MASON scales linearly with increasing proxies, we expect to be able to achieve over 142 Mops/s before the sequencer becomes the bottleneck. At that point, we expect to be able to continue doubling the ordering throughput of MASON by doubling the number of proxies and doubling the batch sizes. Average batch size for 48 proxies with one sequence space is ~8 operations.

3.5.2 Making CORFU Scalable

MASON provides scalable ordering that, when coupled with a scalable execution protocol, enables services to scale. Corfu-MASON replaces CORFU's monolithic sequencer with MASON, yielding a scalable distributed shared log (§3.4.2).



(a) `append` throughput-latency

(b) `read` throughput-latency

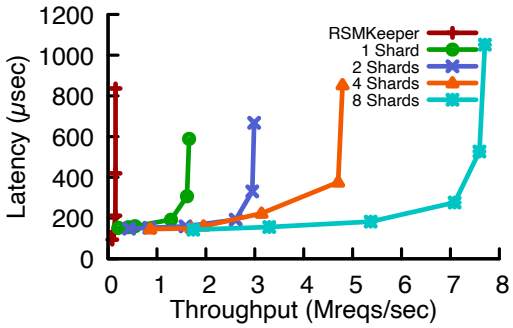
Operation	Med.	99%
CORFU′ `append`	70	90
Corfu-MASON `append`	200	297
CORFU′ `read`	32	62
Corfu-MASON `read`	97	147

(c) Latency (μs).

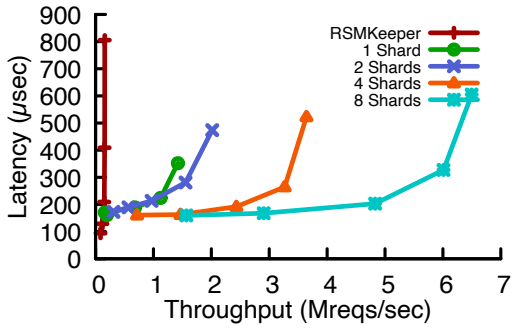
Figure 3.7: CORFU′ and Corfu-MASON comparison; each point represents a given load, doubling the client load from the previous point. Corfu-MASON `append` throughput scales linearly with more shards while CORFU′ saturates at 2 shards. Corfu-MASON has higher latency in exchange for contiguity and linear scalability.

We compare Corfu-MASON with CORFU′, our implementation of CORFU in the same environment as Corfu-MASON, using C++ and eRPC over DPDK. CORFU′'s sequencer processes requests at ~14.2 Mops/s, nearly line-rate for our message size (~14.5 Mops/s). This is a fairer baseline than using CORFU's improved sequencer, whose maximum ordering throughput is ~570 Kops/s [5, 6].

Figure 3.7(a) evaluates Corfu-MASON's scalability. We run a workload consisting entirely of 64 B appends and increase the number of Corfu shards. We use 6 (replicated) proxies for every Corfu shard, keeping the ratio of proxies to Corfu shards constant. CORFU′ roughly doubles throughput from one to two Corfu shards before the sequencer saturates and latency increases; the maximum observed throughput of CORFU′ is ~14.1 Mops/s with latency of ~70 (median)/~90 μs (99th percentile). MASON allows ordering in Corfu-MASON to scale, enabling service throughput to increase linearly: Corfu-MASON scales from ~7.3 Mops/s with one Corfu shard to ~29.1 Mops/s with four Corfu shards, an increase



(a) `setData` throughput-latency



(b) `getData` throughput-latency

Operation (shards)	Med.	99%
RSMKeeper `set`. (1)	211	355
ZK-MASON `set`. (1)	192	268
ZK-MASON `set`. (8)	276	518
RSMKeeper `get`. (1)	209	352
ZK-MASON `get`. (1)	224	304
ZK-MASON `get`. (8)	327	699

(c) Latency (μs).

Figure 3.8: RSMKeeper and ZK-MASON comparison; each point represents a given load, doubling the client load from the previous point. ZK-MASON achieves higher throughput than RSMKeeper with a single shard at comparable latency. ZK-MASON throughput scales linearly at the cost of a modest increase in latency.

of ~3.98x. Append latency at four Corfu shards is ~200/297 μs. The increase in latency is from extra round trips (clients sending requests to proxy leaders, which leaders replicate) and proxies waiting for 20 μs to batch requests.

Figure 3.7(b) shows the scalability of reads. Clients execute reads on random log positions in CORFU′ by reading a shard's tail replica. Reads in Corfu-MASON are executed by proxy leaders, which read the tail replica. Reads are not sequenced in either service, so reads scale the same in both services. Latency for Corfu-MASON is ~97/~147 μs, ~65 μs higher than CORFU′'s ~32/~62 μs, from the extra round trip through the proxy leader.

3.5.3 Making ZooKeeper Scalable

ZK-MASON is a ZooKeeper-like coordination service [26] (see Sec. 3.4.3). ZK-MASON uses a scalable execution protocol with MASON's scalable ordering to scale the entire service.

To compare ZK-MASON and ZooKeeper we implemented RSMKeeper, a prototype of ZooKeeper over Raft [59]. RSMKeeper has the same operations as ZK-MASON. Both are implemented in C++ with eRPC over DPDK [17, 31]; RSMKeeper uses a single thread. We note that RSMKeeper has much higher throughput than the original ZooKeeper implementation, providing a fairer baseline.

We configured RSMKeeper and ZK-MASON to maximize service throughput while keeping latency low. RSMKeeper is loaded by one client machine running 8 threads. ZK-MASON clients use 16 threads. ZK-MASON uses 2 proxies per shard and 1 client machine per proxy. Each proxy uses 8 threads and each ZK-MASON shard uses 1 thread. This is the minimal setup for a single shard that stresses the shard's throughput. We add more ZK-MASON shards, keeping the ratio of clients and proxies to shards constant. Our ZK-MASON experiments show the scalability of the contiguous multi-sequence abstraction when scaling out the number of shards.

Figure 3.8(a) shows the throughput-latency of `setData` operations. RSMKeeper's (and ZooKeeper's) design uses a single replicated state machine to ensure consistency and thus cannot run with more than one shard; its maximum throughput is ~150 Kops/s. With one shard, ZK-MASON has 8.6× the service throughput of RSMKeeper, at ~1.29 Mops/s while providing latency in a similar range as shown in Figure 3.8(c). ZK-MASON has lower latency than RSMKeeper in Figure 3.8(c), ~192 μs vs ~211 μs, because of where we determined overload to be for RSMKeeper; we chose a point in the throughput-latency curve that increased throughput at the cost of some latency. At lower load and lower throughput settings, RSMKeeper has lower latency than ZK-MASON. For example, RSMKeeper has

~94 μs median latency at ~85 Kops/s and ZK-MASON has ~152 μs at ~212 Kops/s. ZK-MASON's higher single-shard throughput comes from the proxy layer scaling with two (replicated) proxies handling client requests for one ZK-MASON shard. Furthermore, ZK-MASON shards do less work per `setData` operation than RSMKeeper. For each operation, RSMKeeper handles operation execution, one round of client-to-leader communication, two rounds of leader-to-follower communication, and snapshotting Raft state and log compaction to disk. On the other hand, MASON frees the ZK-MASON shard from handling tasks related to ordering and consensus. The shard only handles execution and one round of proxy-to-shard communication. With more resources devoted to execution, one ZK-MASON shard has a higher maximum throughput than RSMKeeper. More importantly, ZK-MASON is able to scale throughput by increasing the number of shards and proxies: with eight shards its throughput scales to ~7.1 Mops/s.

Figure 3.8(b) shows the throughput-latency of `getData` operations. We configured RSMKeeper to replicate `getData` operations to provide the same consistency as ZooKeeper's `sync-getData` construction and ZK-MASON's `getData` operation. RSMKeeper's maximum throughput is ~150 Kops/s with latency ~209 (median)/~352 μs (99th percentile). ZK-MASON's `getData` throughput scales from ~1.1 Mops/s with one shard to ~6 Mops/s with eight shards. Latencies in those runs range from ~224/~304 μs (one shard) to ~327/~699 μs (eight shards). `getData` operations have slightly higher latency than `setData` operations because proxies need to wait for a response from a ZK-MASON replica which must execute all operations ordered before the `getData` before returning to the client, while `setData` can be executed on ZK-MASON shards asynchronously.

3.5.4 MASON Provides a Contiguous Sequence

This experiment validates that MASON provides a contiguous sequence despite component failures. We run MASON with 16 proxies. Each proxy machine hosts either 8 leaders or 8 followers in 8 different proxies for a total of 6 proxy machines (2 leader machines and 4

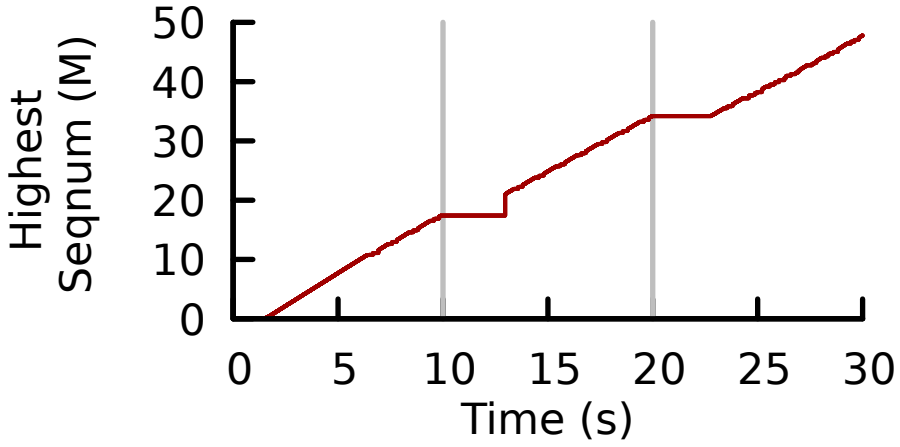


Figure 3.9: Highest contiguous multi-sequence number received across all clients at time *t*. We induce proxy leader failure at 10 s and sequencer failure at 20 s.

follower machines). Load is generated by 4 client machines. Clients request one sequence number from each of 4 sequence spaces. We inject proxy and sequencer failure; network drops occur naturally.

Figure 3.9 shows the highest contiguous sequence number successfully received by a client over time for each of 4 sequence spaces. That is, if Figure 3.9 indicates that at time x the highest contiguous sequence number from a sequence space is y, then each sequence number up to and including y in that space was received by some client. We ran the experiment with 4 sequence spaces and plotted the highest contiguous sequence number for each sequence space. Since clients request one number from every sequence space, they advance at the same rate and thus all four lines overlap.

We first kill a proxy machine hosting 8 proxy leaders 10 s into the experiment. The 8 recovering proxies stop processing client operations and may have uncompleted operations. The flat region in the plot indicates where the sequence increase is blocked by uncompleted operations. Once failover is complete, the new leaders respond to pending client operations. The plot spikes as gaps in the sequence are filled in and operations serviced by the two non-failing proxies are accounted for. Proxy failure detection and failover take 3.06 s, including 1 s-2 s for the failure detection timeout, set randomly by Raft.

We kill the sequencer 20 s into the experiment. A proxy times out 1 s later and begins the recovery protocol. Failure detection and recovery take 2.38 s—the plot's 2nd flat region—and then the contiguous multi-sequence continues to grow.

3.6 Limitations

Resource Cost. MASON provides a contiguous, scalable multi-sequence using replicated proxies. This ability to scale adds overhead compared to non-scalable designs in operational settings that do not demand more throughput than the latter can support. For instance, when CORFU′ saturates its sequencer at 2 Corfu shards, Corfu-MASON uses 36 proxy machines (12 proxy groups). Proxies process more RPCs than Corfu shards, so Corfu-MASON needs more proxies than shards to saturate the shards. The resource overhead (number of machines) is thus 600% to saturate two Corfu shards with Corfu-MASON (42 total machines including a standby for Corfu-MASON and 6 including a standby for CORFU). As another example, the overhead for the single-shard setup for ZK-MASON is 266% (8 total machines including a standby for ZK-MASON and 3 total machines for RSMKeeper).

However, MASON's ability to scale means it can be used to provide throughput that cannot be achieved with non-scalable designs: we evaluated up to a 206% throughput increase for Corfu-MASON over CORFU′'s *maximum* throughput and expect throughput to continue scaling, and we evaluated up to a 4733% throughput increase for ZK-MASON over RSMKeeper's *maximum* throughput and expect throughput to continue scaling.

Thus, a practical deployment strategy could be to initially deploy the service in a small one proxy setup and to colocate the service's processes on the machines used for the proxy. As throughput demands increase, the service could then add more proxy groups and eventually split proxies and the service's processes into different machines to scale them independently. This strategy may add some operational overhead from changing configurations but would enable a service to pay a lower cost for an initial setup.

Performance predictability. The intermediate components between clients and the service can make the performance of the system as a whole less predictable because tail latencies at each hop can accumulate. Furthermore, MASON's additional components may complicate performance debugging since more components will need to be inspected.

3.7 Related Work

This section explains MASON's relationship to the five categories of related work it builds upon. At a high level, the primary distinction of MASON is that it provides strict serializability, unlike atomic multicast; it is scalable, unlike state machine replication and fast ordering systems; it provides multiple sequence spaces, unlike shared logs; and its abstraction enables more efficient, specialized service implementations than distributed databases.

Atomic multicast. Atomic multicast guarantees messages are delivered reliably and satisfying a total order to one or more groups of processes [11, 20, 21, 24]. Unlike the order given by a contiguous multi-sequence, the total order given by atomic multicast is not strictly serializable. Atomic multicast is thus used directly in systems to provide weaker consistency guarantees [48], or augmented to provide stronger consistency [7, 42].

State machine replication. There is a large body of work on state machine replication (SMR) implemented with consensus [1, 15, 18, 23, 27, 29, 32, 35, 36, 38, 41, 49, 51, 55, 58, 59, 63, 73], which provides two properties MASON aims for: a contiguous sequence via SMR's log and fault tolerance via consensus. These protocols have a fundamental throughput ceiling, the rate a single machine can execute commands in order.

Compartmentalization is a technique to scale state machine replication [73]. Compartmentalization "involves decoupling individual bottlenecks into distinct components and scaling these components independently". How MASON scales ordering can be viewed as compartmentalization: ordering, handing out multi-sequence numbers, is explicitly separated from execution, and scaled via the proxy layer and batching.

In the compartmentalized version of Multi-Paxos, a batching layer batches client requests before sending them to the leader which orders batches in the log [73]. This technique scales ordering, but is still limited to the rate a single machine can execute commands in order, and does not easily extend to the contiguous multi-sequencing abstraction.

Distributed shared logs. CORFU uses a monolithic sequencer to find the tail of a distributed shared log [5]. It cannot scale beyond the throughput of the sequencer. MASON can provide a contiguous sequence to a CORFU service while scaling beyond the throughput of a monolithic sequencer, but MASON requires more resources and has higher latency.

Delos [4] unifies separate shared log or storage instances into a single virtualized shared log. It inherits the scalability limitations of its underlying systems. Scalog [16] is a distributed shared log that uses a replicated ordering mechanism to reliably totally order records in a log. Scalog increases the write throughput ceiling compared to CORFU by two orders of magnitude. It increases ordering throughput using a similar technique as MASON's proxies which batch requests across clients: storage servers collect and periodically order multiple operations at once using tiered aggregators "that relay ordering information" from the layer below it up to a replicated sequencer. Scalog and MASON both guarantee that services always see a contiguous sequence of operations, unlike CORFU. Scalog's mechanism for guaranteeing contiguity is similar to MASON's, i.e., both replicate client operations before executing them. Scalog also replicates its sequencer. However, Scalog cannot be easily extended to multi-sequencing: Scalog orders operations using a summary of operations that arrive at individual shards. Scalog's resource overhead is lower than MASON for services where replicating an operation is the same as executing the operation (e.g., a shared log). In such services, Scalog replicates the operation on servers for contiguity, which serves to execute the operation as well. The same service over MASON must replicate the operation twice, as separate steps on distinct components: the proxy layer replicates the operation for contiguity, and execution is carried out by storing the operation on service servers. For other services where executing the operation is more than just storing its input, for example in ZK-MASON, Scalog's technique of replicating for contiguity would need to be accompanied by a separate step of executing the operation. MASON's and Scalog's overheads are thus similar for such services.

ChronoLog [34] uses physical time to order records by accounting for skew among distributed components. It reports an order of magnitude higher throughput than CORFU. Like Scalog, Delos and ChronoLog cannot be easily extended to multi-sequencing: ChronoLog and Delos lack mechanisms to atomically append to multiple logs. Thus, they cannot easily be modified to support strictly serializable cross-shard operations.

Chariots [54] scales by delegating the ordering of disjoint ranges of a shared log to independent servers, providing only causal consistency [47]. FuzzyLog partially orders records in exchange for better performance [48]. MASON provides the stronger guarantee of strict serializability.

Fast ordering systems. State-of-the-art networks or network appliances can support high-throughput, low-latency sequencing [30, 44, 45]. Unlike MASON, these sequencers cannot scale, do not provide a contiguous sequence, and are not fault-tolerant. However, such sequencers can provide sequencing with much lower latency than MASON.

Kronos provides high-throughput happens-before ordering; services totally order operations [19]. Mostly-ordered multicast uses datacenter network properties to provide consistent multicasting except during network failures or packet loss [63]. Reliable 1Pipe, 1Pipe's strongest abstraction, provides ordered communication to receiver groups where messages eventually arrive absent failures and partitions [43]. Services detect and handle lost messages with consensus, much like services using noncontiguous multi-sequences. In contrast to these systems, MASON provides the stronger abstraction of a strictly serializable, contiguous sequence.

Distributed databases. FoundationDB uses a single sequence space with batching to scalably implement commit timestamps [78], but does not provide contiguity or multi-sequencing. Granola uses clock-based timestamps on clients (through "client proxies" which are similar to our service stubs in that they exist on the clients (proxies) to execute Granola (service) code) and servers and coordination among shards to determine a global transaction

order [13]; the clock-based timestamps do not provide contiguity or multi-sequencing. Calvin uses a sequencing layer distributed across all servers in the system [68]. The sequencing layer synchronously batches operations, exchanges them among replicas, then exchanges them among all servers in its copy of the database. Distributing the sequencer across servers and using large synchronous batches enables the sequencer to be more scalable than a single machine sequencer. However, as more shards are added, the all-to-all communication within a copy of the database will become a bottleneck, halting scalability. Eris [44], Calvin [68], vCorfu [71], Tango [6], and other distributed databases [3, 53, 62, 64, 76, 78] provide a higher-level abstraction than MASON. It is harder for services to build efficient, specialized implementations over the distributed database abstraction compared to the multi-sequence abstraction. For instance, ephemeral znodes (§3.4.3) do not fit the traditional distributed database model; a service developer would implement a new replicated component to manage client connections and explicitly delete the znode at connection termination. In contrast, implementing ephemeral znodes in ZK-MASON was straightforward.

The multi-leader approach to system design, as used in, for example, Spanner [12], uses a replica designated as the leader for each shard. A shard leader coordinates with replicas in its own shard and with leaders in other shards for operations that span multiple shards. For example, the service must implement consensus to order operations within a shard, perhaps via state-machine replication, and concurrency control, like optimistic concurrency control or two-phase locking with two-phase commit, to order operations across shards. Thus, multi-leader services are more difficult to implement than services using the multi-sequence abstraction, which orders operations within a shard by assigning a sequence space to a shard and across shards by atomically allocating a multi-sequence number. Therefore, services built on the multi-sequence abstraction do not need to implement coordination across shards for ordering; they only need to implement the service semantics.

MASON's contiguous multi-sequence abstraction is an excellent candidate for implementing distributed databases. Its contiguity would eliminate significant complexity in ported implementations of Eris and vCorfu. Similarly, its contiguity would greatly simplify developing new multi-sequence-based distributed databases. Its scalable multi-sequence would enable Eris, vCorfu, and future databases to scale far higher than the throughput ceiling of monolithic sequencers. This is an important avenue for future work.

Chapter 4

Redefining Slowdown Tolerance

We redefine slowdown tolerance to be achievable in the wide-area, to not rely on arbitrary thresholds, and to be used to compare across protocols.

4.1 Existing Definitions are Insufficient

Slowdown tolerance as defined in Copilot is unachievable over a wide-area network. The definition is based on defining a slow replica and replacing a slow replica with the slowest of the fast replicas [55]. Specifically, a replica is defined to be *slow* "when its responses to messages take more than a threshold time t over its normal response time." Slowdown tolerance is then defined: "An RSM is *s-slowdown-tolerant* if it provides a service that is not slow despite s replicas being slow. More specifically, sort the replicas $\{r_1, ..., r_s, ..., r_n\}$ of an RSM according to the current definition of slow, such that $\{r_1, ..., r_s\}$ are the s slowest replicas. Let T represent how slow the RSM is—i.e., its response time properties based on the current definition of "slow"—and let T' represent how slow the RSM would be if replicas $\{r_1, \ldots, r_s\}$ were all replaced by clones of $r_s + 1$. An RSM is s-slowdown-tolerant if the difference between T and T' is close to zero."

The Copilot definition thus compares the latency of the RSM in the slowdown case, T, to the hypothetical case where the slow replicas have latency equivalent to the slowest

fast replica, T'. In the wide-area, due to latency heterogeneity it could be impossible to ensure that $T - T'$ is close to zero. Consider the following illustrative example: there are three replicas r_1, r_2, and r_3. r_1 has one-way message delay of 100 ms to both r_1 and r_2. r_1 and r_2 have 200 ms delay to each other. Clients are colocated with r_1. When r_1 becomes slow the Copilot definition produces T' by cloning r_1 with r_2 (or r_3 which is equivalent). Clients now have 100 ms delay to r_1, which now has 0 ms delay to r_2 and 200 ms delay to r_3. T' is thus 200 ms: clients send commands to r_1 (100 ms), r_1 can receive a quorum with r_2 (0 ms), and reply to clients (100 ms). T, when r_1 is indefinitely slow is at least 600 ms: clients send commands to r_2 (100 ms), r_2 receives a quorum with r_3 (400 ms), and replies to clients (200 ms). We note that this example is general: the lower bound for consensus is 2 message delays when there are no conflicts [33].This illustrates why achieving the Copilot definition of slowdown tolerance in a WAN is not possible.

The definition of a "slow" replica is based on an arbitrary threshold t. Designing and implementing a slowdown tolerant protocol thus requires reasoning about and instantiating thresholds which can be difficult as a reasonable choice could rely on different factors: latency properties of the network, how fast a replica can normally process messages, and business-level decisions. For example, Copilot used thresholds to bound its performance to be within 10 ms [55]; however, in a WAN, where latencies are an order of magnitude higher, the thresholds may need to be much higher: 10 ms latency variation may be normal.

Copilot's definition is also difficult to use to compare slowdown tolerance across protocols. The definition is binary: either a protocol is slowdown tolerant for a given threshold or it is not. One mechanism that could be used is the chosen threshold, but these can vary depending on the setting the protocol is designed for. In the same setting comparing protocols would rely on the protocols using the same definition of "slow", the threshold t.

We thus redefine slowdown tolerance to enable slowdown tolerance in a WAN, not rely on thresholds, and enable comparison across protocols.

4.2 Slowdown Tolerance Definition

We model the system as a set of processes communicating over point-to-point links. Processes may fail according to the crash failure model, where processes stop executing requests, and the failure is undetectable to other processes. We assume an asynchronous network model where messages can be arbitrarily delayed and reordered.

We define *slowdown tolerance* to be:

An RSM is ε-*slowdown-tolerant* to the slowdown of s nodes if it is guaranteed to respond within ε of the same RSM when any s nodes are removed.

We consider one slowdown to be the slowdown of any n links that share one endpoint.

This definition of slowdown tolerance is applicable both to the local-area, where latencies are homogeneous and small, and the wide-area, where latencies are heterogeneous and large. Thus, this definition is still compatible with protocols designed for slowdown tolerance in a local-area network. It does not rely on defining a "slow" replica and thus does not require defining or reasoning about a threshold. Because the definition does not rely on the protocol itself deciding on a threshold and because it is not a binary definition it enables comparison across protocols.

Table 4.1 shows the latency of protocols when there are no slowdowns and their ε categorization for $s = 1$. For this categorization we present a simplified scenario where replicas are equidistant from each other with latency D, and clients are colocated with replicas. TO represents a configurable timeout. Multi-Paxos is the current state-of-practice and can commit and return to clients local to the leader in one wide-area round trip; clients that are further away have another round-trip to the reach the leader [35]. EPaxos improves normal case latency: when there are no conflicts EPaxos can commit local clients' requests and return in one wide-area round-trip [51]. Both Multi-Paxos and EPaxos are slow when the leader, or local leader, is slow respectively.

System	Normal Case w/ leader(s)	Normal Case w/ others	ε
Multi-Paxos	2D	4D	∞
EPaxos	[**2D**,4D]	[**2D**,4D]	∞
Aardvark*	3D	4D	3TO+2D
Copilot	[2D,**4D**]	[4D,**6D**]	TO+4D
Copilot-Ping-Pong	[**2D,4D**]	[**4D,6D**]	TO+6D
Latent Copilot	2D	4D	TO+6D
AVICENNA	2D	4D	6D

Table 4.1: Normal case latency and 1-slowdown tolerance comparison. For simplicity, we consider a setting with five replicas equidistant from each other and clients colocated with each replica. "D" indicates one message delay. "TO" indicates a configurable timeout. We separate out latency for clients colocated with a leader, or either of copilot's two pilots, and latency for clients colocated with other replicas. Where ranges are given common cases are in bold text. Protocols marked * are byzantine fault tolerant.

Aardvark is a byzantine fault tolerant system that has a leader change mechanism to prevent a byzantine leader from continuously slowing down the protocol [2]. Clients send requests to a leader and all replicas can learn of the commit in three message delays. Thus, clients close to the leader will have three message delay latency, where clients far from the leader will have an extra message delay to send the request to the leader. The normal case latency for a client colocated with the leader is thus higher than Multi-Paxos, comparable to Copilot-Ping-Pong (described below). The worst slowdown case is when the client request is slow to the leader the client will timeout and resend the request to all replicas. The replicas then wait for the leader to propose that client request; the wait is upperbounded by another timeout which needs to be triggered twice. It then initiates leader change which takes 3D to complete. The new leader can begin the protocol to commit during the last message and commit in 3D so that leader change and commit of the client's request takes 5D. This is then 3TO+7D total and if the replica was removed it would complete the same client's request in 5D; so ε is 3TO+2D. It is trivial to make this 2TO+2D by having clients send requests to all replicas immediately rather than timing out waiting for the leader.

Copilot is Copilot without the ping-pong batching optimization and Copilot-Ping-Pong includes the optimization. Copilot, Copilot-Ping-Pong, and Latent Copilot can bound ε [55]. They do this by having two leaders propose and resolving conflicts by executing a round of Basic Paxos to commit the other leader's entry. Note that though ε is 4D+TO for Copilot it's total latency during a slowdown would be at most 10D+TO which is the same as Latent Copilot and AVICENNA.

Normal case vs. slowdown case latency tradeoff discussion We can see that there is no prior protocol that provides competitive WAN latency and any slowdown tolerance, except for Latent Copilot. However, Latent Copilot is complex, requires thresholds to be set (as a result of being built on Copilot), and doesn't accurately detect and appropriately react to all slowdowns. Based on the lessons learned from Latent Copilot we build AVICENNA.

Chapter 5

Avicenna

5.1 Reactive versus Proactive

This section motivates reactive slowdown tolerant designs and an examination of the first design of reactive slowdown tolerance with competitive latency in a wide-area network, Latent Copilot.

5.1.1 Why Reactive Slowdown Tolerance

Proactive slowdown tolerant protocols create disjoint paths through which a client's command could be ordered [55, 60]. For example, a protocol could be designed so that any replica could order a client command at any given time. The intuition is that because there are more than s paths that can order a client command, if there are s slowdowns there is still at least one path that can order commands.

We will describe Copilot's 1-slowdown tolerant design for clarity of exposition though the arguments generalize to s-slowdown tolerant designs. Copilot uses two designated replicas, called pilots, that are both able to order client commands. Each pilot owns a log and all replicas, including both pilots, maintain both logs. Replicas merge the two pilots' logs based on each entry's dependency that states the latest log entry from the other pilot

that should be applied before it. Dependencies are thus transitive and explicitly maintaining one dependency for each log entry contains enough information to order the commands in the entry. Dependencies are established during the time of ordering and used to execute the commands later.

Copilot can order commands in two message delays, the fast path, by establishing initial dependencies during the PreAccept phase. It sets the initial dependency to be the latest log entry seen of the other pilot. If that is also the latest entry seen by the other replicas they will reply to the PreAccept message with a PreAcceptOK message. If a replica has a more recent entry from the other pilot, then there is an ordering conflict: one pilot has PreAccepted later entries the PreAccepting pilot is unaware of, and thus they are ordered before the PreAccepting entry on one pilot and after the PreAccepting entry on the PreAccepting pilot.

To prevent conflicts from hurting normal case latency, Copilot includes an optimization called ping-pong batching where pilots alternate PreAccepts: one pilot waits for the other's PreAccept before sending its own PreAccept. This ensures that each pilot's latest seen dependency is the correct dependency and so each pilot can commit on the fast path.

However, there are two results from this optimization: first, a pilot now has to react to when the other pilot is slow, meaning Copilot with ping-pong batching is a reactive protocol, not proactive; second, clients local to a pilot have to wait up to a full wide-area round trip for the pilot to send a PreAccept for the client's command, for example if the pilot has just sent a PreAccept. Furthermore, because the lower bound of consensus when there are conflicts is 3 message delays (except for some specific cases) and the lower bound without conflicts is 2, there is a fundamental normal case cost that a proactive protocol will have to pay [33, 40].

Because the cost of a proactive protocol under conflicts is 3 message delays and because avoiding that cost results in a reactive protocol, we argue for a reactive slowdown tolerant protocol design.

5.1.2 Latent Copilot

Latent Copilot was the first attempt to provide reactive slowdown tolerance and has competitive latency to the state-of-practice, Multi-Paxos, in a wide-area network [56].

Similar to Copilot, Latent Copilot consists of two pilots an active pilot and a latent pilot. A latent pilot starts a timeout when it receives a client command, if it has not received a commit from the active pilot before its timeout fires, it then sends a PreAccept for that client's command. Replicas reply to the latent pilot and indicate if they have already committed this command. If all replies indicate they have not committed, then the latent pilot executes a "fast takeover" of all preceding commands that are not committed and commits its entry.

The fast takeover protocol is the same as the "fast takeover" protocol of Copilot: the Propose and Accept phases of Basic Paxos [35]. Once it does that *min-new-proposals* consecutive times, the latent pilot promotes itself to be an active pilot. It remains an active pilot for at least *min-active-window* time. An active pilot checks to see if it should demote itself to be a latent pilot by comparing when it commits an entry to when it learns of an entry being committed by the other pilot. If its commit is slower than when it learns of the other pilot's comments for *min-new-commits* times, it demotes itself to be a latent pilot. *min-new-proposals*, *min-active-window*, and *min-new-commits* are all configurable parameters to increase the stability of the protocol so that it does not change active/latent pilots very frequently hurting normal case latency.

Latent Copilot provides competitive wide-area latency to the state-of-practice and has similar slowdown tolerance to Copilot but has several issues. First, it does not accurately detect all slowdowns, for example if the link between the active and latent pilot is slow, in both directions, both pilots will think the other pilot is slow. Second, Latent Copilot requires tuning many knobs: setting thresholds for timeouts, *min-new-proposals*, *min-active-window*, and *min-new-commits*, are all difficult, what they should be is not clear, and the choice will depend on many factors. Latent Copilot is also complex: it inherits some complexity from

Copilot including fast and slow quorums, tracking dependencies to merge two logs, and fast takeover.

In summary, Latent Copilot has competitive normal case latency to the state-of-practice, Multi-Paxos, in a wide-area network enabled by the choice of a reactive instead of a proactive protocol and has some level of slowdown tolerance. However, its detection mechanism, namely timeouts, is insufficient to correctly detect all types of slowdowns, it requires tuning many knobs, and is complex. We instead aim to design a reactive slowdown tolerant protocol from the ground up.

5.2 Design Overview

This section describes the design goals of AVICENNA and each component of AVICENNA and its purpose. We design AVICENNA based on the following design goals:

1. AVICENNA should have latency comparable to the state-of-practice: leader-based protocols like Multi-Paxos and Raft in the wide-area.

2. AVICENNA should be able to quickly change leaders.

3. AVICENNA should be able to quickly detect when another leader is better.

To achieve (1) AVICENNA is a leader-based protocol where clients send messages to every replica, the leader sends an Accept message to every replica, and upon receiving a simple majority quorum of $f + 1$ AcceptOk responses can return to the client. Thus, like Multi-Paxos, AVICENNA can commit in two message delays (excluding the client sending the command) in the absence of leader changes, the lower bound for consensus [33].

To achieve (2), AVICENNA uses a fast leader change protocol and introduces the *delegate* role that enables it to reach a leader-change quorum faster. AVICENNA's leader change protocol is inspired by MR99, a single-instance multi-value consensus protocol that uses an unreliable failure detector [10, 52]. MR99 uses a deterministic leader order to enable a leader change protocol that rotates leaders within one message from the closest quorum to the next leader. AVICENNA designates a single replica for each leader a *delegate*. The delegate can use extra leader change votes on behalf of far nodes so that the next leader receives a rotate quorum faster than waiting for far nodes to send rotate votes. AVICENNA chooses the delegate to be the replica that minimizes the delay from the current leader to the delegate plus the delay from the delegate to the next leader. To ensure correctness, far nodes promise never to reply AcceptOk, but may vote to rotate, which will be treated as duplicate votes if the delegate has already voted.

To achieve (3) rather than attempting to detect a slowdown, which is difficult (§5.1.2), AVICENNA instead detects when the next leader is a better choice. To this end AVICENNA explicitly evaluates if another replica would be a better leader. By asking this counterfactual question, and explicitly answering it, AVICENNA is able to avoid the complexity of accurately detecting slowdowns, which previous protocols are unable to do. AVICENNA uses another replica, acting as a *ghost leader*, to evaluate if it would be better. The ghost leader executes all of the ordering work that the leader executes and clients track real latencies and ghost latencies that AVICENNA uses to decide which replica is the better leader. To decide which is "better" is defined by an *objective function* that the application can define. To reduce the throughput and resource utilization overhead of running the same protocol on the same set of replicas twice, applications can determine which client commands are process through the ghost protocol, explicitly trading off how fast AVICENNA can react to slowdowns and overhead.

5.3 Design

This section describes the design details of AVICENNA.

5.3.1 Normal Case Protocol

Like other replicated state machine protocols, each replica maintains a log of entries. Each entry contains a batch of client commands batched over a specified amount of time. For ease of explanation we describe AVICENNA without batching, and describe the batching mechanism at the end of this section, §5.3.4. When an entry and all preceding entries in the log are committed the replica can execute the commands in the order of the log. Every replica executes commands and returns to the client.

Each replica maintains a list of *configurations* where each configuration maps replicas to *roles*, for example in a given configuration one replica is the leader and another is an acceptor. The list of configurations is the same on each Replica. Each replica also maintains p, a phase number, that represents the current configuration. When a replica increments p its active role changes, for example from acceptor to leader. The next configuration is thus deterministic.

To have normal-case latency comparable to the state-of-practice, AVICENNA is a leader-based protocol. When a replica receives a command from a client, if its current configuration indicates it is the current leader it sends an Accept message to every replica containing the command to Accept, the log position of the entry, and its current phase, p. If the replica is in the same phase and is an acceptor it replies AcceptOk. Upon receiving a simple majority quorum of $f+1$ votes, where there are $2f+1$ replicas, the leader commits the entry and sends a Commit message to every replica.

Thus, in the normal case AVICENNA has the same number of message delays before committing a client's command as other leader based protocols like Multi-Paxos and Raft [35, 59].

5.3.2 Slowdown Reaction Protocol

A reactive protocol can be separated into two components – detection and reaction. We first describe AVICENNA's reaction protocol.

To react to the detection of a better leader AVICENNA changes configurations through a protocol inspired by MR99 that allows the next leader to begin sending Accepts after receiving a Rotate message from either one replica or the closest simple majority [52].

Each replica, for each entry i in the log, maintains c_i the command for which it will send an Accept if it becomes the leader. c_i is initially set to any client's command it received. Upon receiving an Accept from a leader in the same phase with command a_i replicas set c_i to a_i. This is so that if a_i is committed by the leader it will exist in any quorum. Using this property, AVICENNA's reaction protocol guarantees that the next leader will either commit a_i or send an Accept for a_i.

Leader-change protocol. Upon receiving a Rotate message or when the replica's local slowdown detector suspects the next leader is better the replica sends a Rotate message that contains p and its log $\{a_i, ..., a_j\}$ of commands for which it received an Accept from the leader in phase p where a_i is the command in log entry i and i is the latest entry where all entries $i-1$ and below have been committed. j is the latest entry for which it has received an Accept from the leader in phase p. For an entry k where $i < k < j$ and the replica has not received an Accept for k it sends an empty value indicating it did not receive an Accept for that entry. The Rotate message implicitly contains an empty command for every entry later than j. Once a replica sends a Rotate message for phase p it ignores all future Accept messages from the leader in phase p.

When a replica receives $f+1$ Rotate votes in phase p it finds the minimum i, I, and the maximum j, J. It iterates through each log entry from I to J and if the command was accepted by a majority it commits the command immediately and broadcasts a commit message. If there was an accepted command in entry k, a_k, it sets its local c_k to a_k. Because

there is only one leader in each phase, p, sending Accepts for each command, any replica is guaranteed to have received either no command for entry k in phase p or the same command for entry k in phase p. Therefore, it is guaranteed that there is only one such accepted command a_k in any set of Rotate messages in phase p. If no command was received by any replica for entry k in the Rotate quorum then no action is needed.

If a replica receives a Rotate message with an i greater than its own, then the other replica knows about commits that it does not. Thus, it may not receive a quorum of commands in entry i. The replica requests either the committed commands, which it can commit upon receiving them, or a quorum of Rotate messages containing the entry. As an optimization replicas can include committed entries less than their respective is to reduce the likelihood more message delays are needed. Because AVICENNA is designed for the wide-area where extra delays are costly and rotating should be infrequent, the benefit of sending extra committed entries may be worth the cost.

After updating each c_k for every k from I to J the replica increments p and proceeds to the next configuration. If the replica becomes the leader it sends an Accept for every entry from I to J that has not been committed.

Safety. If a command a is committed in entry k, AVICENNA ensures that the only possible value that any future leader will send an Accept for in entry k is a. If a was committed for entry k by the leader for phase p, then the command was received by a quorum of replicas in phase p. Thus, during rotation, any quorum will contain at least one replica that indicates that it received a from the leader for phase p. Any replica that proceeds to phase $p+1$ will have received a quorum of Rotate messages where at least one message contains a in entry k and will set its corresponding c_k to a. Therefore, all replicas in phase $p+1$ will have their c_k equal to a, so, any replica that becomes the leader will send an Accept for a in entry k.

Delegates. In a wide-area network, because some replicas are farther from the current leader than other replicas they are unlikely to participate in normal case Accept quorums.

Because of this AVICENNA designates a replica in each configuration the *delegate* and $f-1$ replicas as *standbys*. The delegate is allowed to vote to Rotate on behalf of standby replicas with all empty entries. Standby replicas are not allowed to participate in Accept quorums but may send Rotate with empty entries, receive commit messages, execute, and return to clients. Because the delegate has f Rotate votes (itself and $f-1$ standbys) the next leader can complete rotation and begin proposing after hearing only from the delegate (and itself).

Safety is still guaranteed. Duplicate Rotate votes from a standby replica and the delegate are filtered, i.e. a Rotate vote from a standby either through the delegate or the standby is only counted once. In all other aspects a standby replica behaves as if all Accept messages to it are dropped.

For each configuration we choose the $f-1$ replicas furthest from the leader to be standbys and the delegate to be the replica that minimizes the delay from the leader to the delegate plus the delay from the delegate to the next leader; this may be the next leader itself, in which case it similarly only needs to hear from one other replica to Rotate. The intuition is that this is the replica mostly likely to be fastest to detect a slowdown on the leader and notify the next leader.

Because $f-1$ replicas, the standbys, cannot vote to Accept, the failure of two replicas that are not standbys can then cause the leader to not be able to reach an Accept quorum because there are f acceptors left. Furthermore, it is possible where there is no configuration that has enough acceptors. To handle this case AVICENNA includes *Armageddon* configurations where there are no standbys to ensure that progress is made. To ensure safety, which phase the Armageddon configuration occurs in must be deterministic and AVICENNA needs an Armageddon configuration with a leader that has not failed. So, a simple choice would be including $f+1$ Armageddon configurations, each with a different leader so that one is ensured to be alive, every A phases where A is configurable at the beginning of the protocol. A should be large enough to allow multiple rotations of the protocol before entering an Armageddon configuration to handle transient slowdowns. Furthermore, when the phase is

close to an Armageddon configuration, but progress can be made, replicas can rapidly vote to Rotate until the current phase passes the Armageddon configurations.

5.3.3 Slowdown Detection

AVICENNA is designed to tolerate one slowdown. AVICENNA's slowdown detection protocol is based on explicitly evaluating the counter factual question of what the latency would be if AVICENNA rotated to the next configuration.

Ghost leader for counterfactual evaluation. AVICENNA uses a *ghost leader* that executes the protocol the same way the real leader executes the protocol except for execution, replicas do not execute commands committed via the ghost protocol. Clients and replicas receive real and ghost messages. The latency that clients see through the ghost protocol is used by replicas to determine when the next leader is better than the current leader. A slowdown is thus not explicitly detected; the replicas always try to choose the best leader. AVICENNA uses an *objective function* that is defined by the application. For example, an objective function could minimize the average latency seen across all clients, minimize the minimum latency seen by any client, or minimize the maximum latency seen by any client. Client latencies are not given to the objective function in the same order on every replica.

Ghost protocol. To calculate the latency that client commands would experience through the ghost leader AVICENNA uses a series of messages that mimic the real protocol and clients measure the latency experienced. Clients send commands to every replica. The real leader sends Accepts and if the command indicates AVICENNA should also process this command through the ghost protocol, the ghost leader sends GhostAccepts. Replicas reply with AcceptOks and GhostAcceptOks. The real leader and the ghost leader also send GhostAcceptOks and AcceptOks to the ghost leader and the real leader respectively. When a replica learns that an entry is committed via the real protocol or the ghost protocol, it sends a RealCommitted or GhostCommitted message to the client respectively. Note that

the RealCommit is not the reply to the client, the command may still need to be executed to send the Reply.

When replicas send Rotate messages and execute the rotation protocol the ghost log is kept up-to-date in the same way as the real log.

The client sends RealCommitLatency and GhostCommitLatency messages to each replica. If the client receives a GhostCommitted message but has not received the corresponding RealCommitted message, the client waits for a period and then begins sending RealCommitAtLeast messages that contain the latency of the RealCommitted message if it were to arrive when sending the RealCommitAtLeast message. The client then sends RealCommitAtLeast periodically. In our implementation we choose a period equal to the batching period of replicas (§5.3.4).

RealCommitted, GhostCommitted, and RealCommitAtLeast messages are sufficient to react to a slowdown that occurs on the real leader before it was committed in the real log. The latency that is left for the real protocol is execution, which begins in parallel to sending the RealCommitted message, and sending the Reply to the client. The latency that *would* be left for the ghost protocol would be execution, which begins in parallel to sending the GhostCommitted message, and sending the Reply to the client. Thus, if neither the real protocol nor the ghost protocol is slow the client would expect to receive a Reply (or a GhostReply if one existed) after the time it takes to execute the operation. Thus, GhostCommitLatency and RealCommitLatency can be compared and if GhostCommitLatency is better than either RealCommitLatency or RealCommitAtLeast according to the objective function, then the replica votes to rotate.

All replicas execute, measure how long it took to execute and send a Reply to the client with the result of the command and the execution time. Clients send RealLatency messages to all replicas with the end-to-end latency that it saw and the execution time. As an optimization clients can piggyback RealLatency messages onto the next command message. Replicas use the objective function to compare RealLatency and GhostCommit+execution

time. This will react to slowdowns occurring on the real leader after sending commit messages.

What is not yet detected is slowdowns after sending the RealCommit to the client and before sending commit to the replicas and executing. To solve this the real leader includes the entry's position in the log in the RealCommit message. The client then forwards this commit message to every replica, which can commit, execute, and send a Reply to the client. The client then sends this end-to-end latency in a RealLatency message to every replica.

5.3.4 Miscellaneous

Batching. AVICENNA batches by each replica waiting for a period of time collecting client requests and placing them into a single entry. Both the real leader and ghost leader batch. During rotation the batch of commands that was received during the Accept is sent and rotation proceeds as described above.

Exchanging client commands during rotation. To commit commands earlier when possible, during rotation, replicas will inform other replicas of outstanding commands that they have so that the new leader can send an Accept for them. To do this, instead of sending no command for an entry k, replicas send their own respective c_ks, the value they will propose when they become the leader, indicating this is not an accepted command from the leader. If no replica in the next leader's Rotate quorum received an Accept for entry k from the previous leader and if the next leader has never changed its own c_k from its initial value, then the next leader batches all of the c_ks together and sets c_k to the batch of commands. It is important that the next leader only does this if it never changed its own c_k. If it modified c_k as a result of an Accept or a Rotate then the accepted value may have been committed, and so it must only propose that value. This optimization can result in committing the same command in multiple log positions, so, AVICENNA executes the command in the earliest log entry and ignores duplicate commands that are later in the log.

Choosing configurations. To set the list of configurations, every replica pings every other replica to create a table of latencies which are given to the application to use to choose the best configuration according to its own goals.

Command IDs Because clients send commands to every replica, the content of the command does not need to be sent with real or ghost protocol messages. Instead each client attaches a unique *client_id* and *command_id* to each command. Replicas create a local map of *(client_id, command_id)* to command, and exchange only the *(client_id, command_id)* pairs when executing the protocol. Importantly replicas can only reply AcceptOk if they currently have the command because it is possible for the client to fail and the leader to commit and then fail while being the only replica to have received the client command. To handle this replicas can request the command from other replicas. When there are no slowdowns it would be expected that a replica would receive the client command before the Accept for that command as it should arrive in one delay from the client to the replica as opposed to a delay from the client to the leader plus a delay from the leader to the replica. Small commands that would not cause much overhead can be piggybacked on protocol messages as in other protocols.

5.4 Evaluation

The goal of AVICENNA is to provide slowdown tolerance with latency comparable to the state-of-practice in a wide-area network. In this section we evaluate AVICENNA under long-lived slowdowns of various severities.

Implementation AVICENNA is implemented in Go in the same codebase as Copilot, its variations, and a version of Multi-Paxos to enable apples-to-apples comparisons.

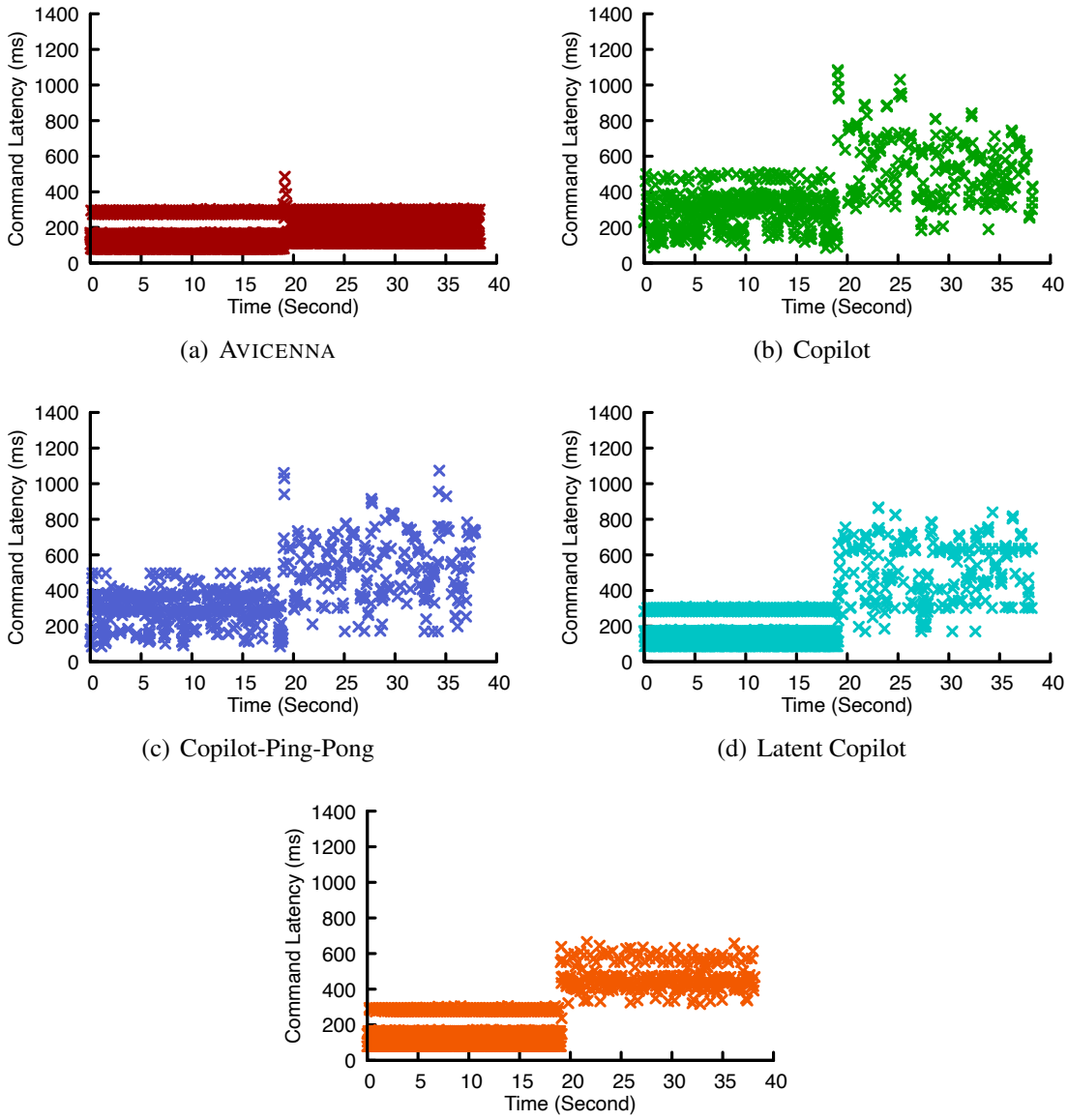
Configuration All experiments are run on Microsoft Azure Standard D8s v3 (8 vCPUs, 32 GiB memory) virtual machines. We run 7 replicas ($f = 3$) in the following locations: East US 2, South Central US, West US 2, Australia East, Southeast Asia, West Europe, and UK South. Each location has one replica and one client.

Baselines We run AVICENNA and all baselines with a batch interval of 10 ms. We configure AVICENNA with an objective function that minimizes maximum client latency. Because even when there are no slowdowns latency can vary by up to the batching interval and there is a small amount of jitter, the objective function only indicates to AVICENNA to rotate if the improvement is 10 ms (the batch interval) plus 1% of the maximum client latency. Furthermore, if the maximum latency is within 5 ms for both the real and ghost leader the objective function next chooses the leader with the minimum client latency and rotates only if the ghost leader has a minimum client latency at least 10 ms lower than the real leader. At the beginning of a run AVICENNA replicas ping each other and choose the best two configurations that minimizes the maximum client latency and then the minimum client latency when the max is within 5 ms. For the leader-based baselines we manually use the best configuration found through this method, and for Copilot and variations we use the best two configurations.

We compare against three versions of Copilot: one version without the ping-pong batching optimization, but with all other optimizations, which we call Copilot, one with the ping-pong batching optimization, which we call Copilot-Ping-Pong, and Latent Copilot [55, 56]. Copilot represents a purely proactive protocol that can commit in two message delays when there are no conflicts, but can commit in 4 message delays when there are conflicts, which we expect to be the common case. Copilot-Ping-Pong ensures the common case is 2 message delays, but causes some client commands to have to wait for up to a full RTT to be PreAccepted. Latent Copilot should have similar latency to AVICENNA and Multi-Paxos in the normal case, but will not be able to react properly to different types of slowdowns. For Copilot, Copilot-Ping-Pong, and Latent Copilot we use a fast-takeover timeout of 136 ms: the expected time to have received a commit plus 100 ms. For Copilot-Ping-Pong we use a ping-pong-wait timeout of 38 ms: one round-trip time between the two pilots plus 10 ms.

We separate Copilot and Copilot-Ping-Pong to show that our intuition is correct that there will be prohibitively many dependencies in the wide-area and that removing these with ping-pong batching is bad for latency compared to leader-based protocols.

To represent the state-of-practice, leader-based protocols, we compare AVICENNA to a *fast view change* version of Multi-Paxos, Multi-Paxos-Fast-View-Change [35]. Multi-Paxos-Fast-View-Change is Multi-Paxos with a deterministic next leader and a low view change timeout. We preconfigure the next best leader determined by AVICENNA's pinging at the start of the experiments, which is also the second pilot in Copilot and its variations. Deterministically choosing the next leader and keeping a low timeout is accomplished via a master that does not become slow and pings the leader every 10 ms; this choice mirrors the choice of batch interval. When 136 ms passes without receiving a response to a ping the master initiates the deterministic view change. We choose 136 ms to mirror the takeover timeout of Copilot and its variations. This allows Multi-Paxos-Fast-View-Change to react quickly to slowdowns on the leader. We compare against Multi-Paxos-Fast-View-Change instead of Multi-Paxos to provide an upper bound for Multi-Paxos's performance.



(a) AVICENNA

(b) Copilot

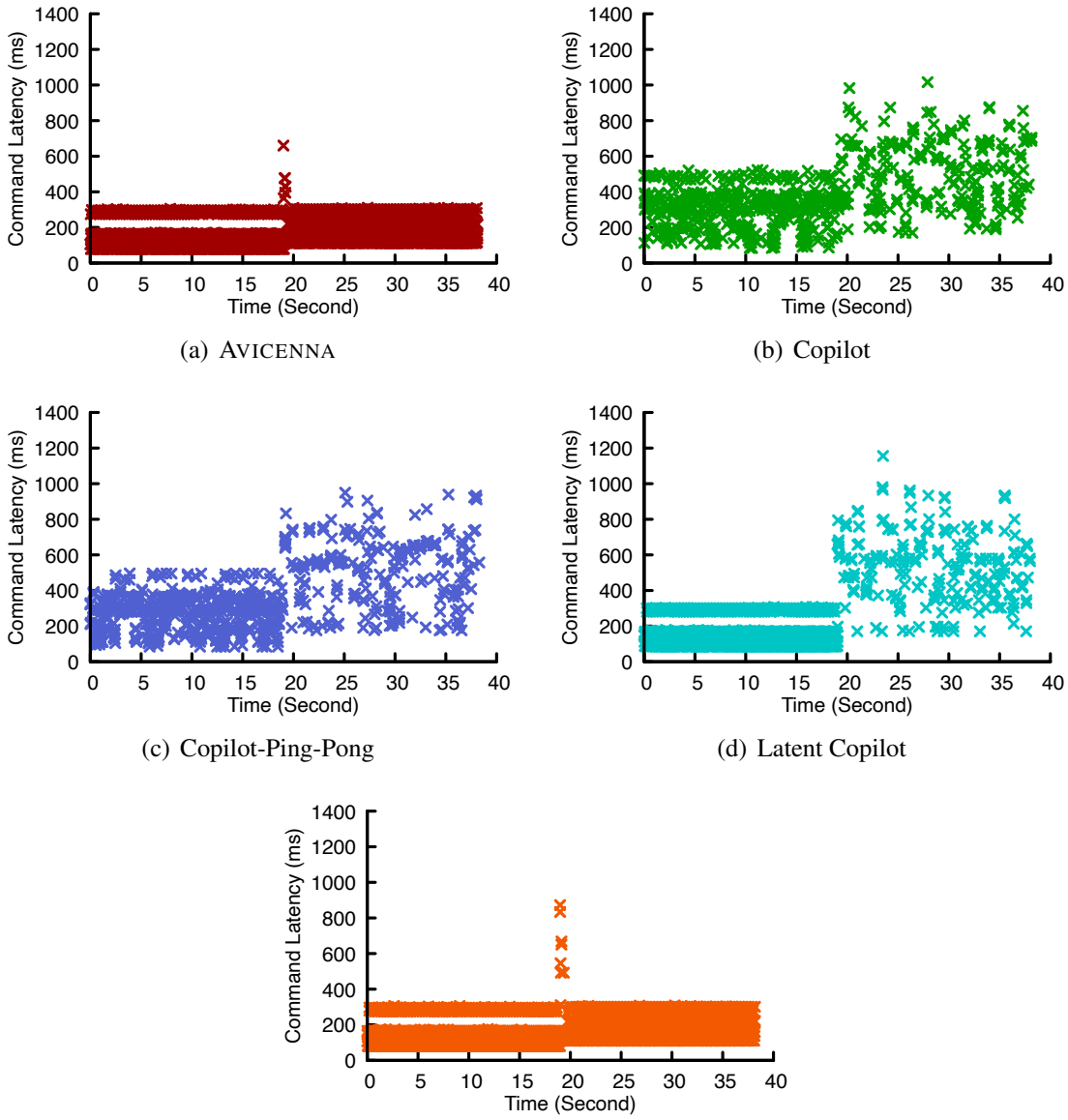
(c) Copilot-Ping-Pong

(d) Latent Copilot

(e) Multi-Paxos-Fast-View-Change

Figure 5.1: Each protocol under a long-lived 68 ms slowdown.

Long-lived slowdowns. To evaluate AVICENNA and the baselines under long-lived slowdowns we inject a slowdown on the leader, or on a pilot for Copilot and Copilot-Ping-Pong, so that the leader sleeps for a period of time before processing a message. We begin injecting slowdowns so that the slowdown begins roughly 25 s into a 50 s experiment. We remove the first 5 s and last 5 s from each plot and plot the end-to-end latency for each request. Load is



(a) AVICENNA

(b) Copilot

(c) Copilot-Ping-Pong

(d) Latent Copilot

(e) Multi-Paxos-Fast-View-Change

Figure 5.2: Each protocol under a long-lived 204 ms slowdown.

driven by closed-loop clients each colocated with a replica. This keeps load low enough to not stress any protocol.

Figure 5.1 shows each protocol under a long-lived slowdown of 68 ms. We choose 68 ms to show a slowdown of a severity half of the configured 136 ms timeout for protocols other than AVICENNA. Figure 5.2 shows the same protocols under a severity of 204 ms, 1.5 x the configured timeout.

Before the slowdown occurs AVICENNA, Latent Copilot, and Multi-Paxos-Fast-View-Change all have similar client latencies because they each use the same leader and commit and reply to the client in the same number of message delays. Copilot and Copilot-Ping-Pong need to resolve dependencies or wait because of ping-pong batching and so have higher latencies.

For both the 68 ms and 204 ms slowdown severities, AVICENNA detects that the ghost leader has a lower maximum client latency than the current leader and rotates leaders, stabilizing latency. Because the next leader has a quorum with a higher latency, requests will have higher end-to-end latency. We see this with all protocols. All versions of Copilot have higher latencies after the slowdown is induced. We believe that because the slowdown is injected for every message on the pilot before the pilot processes that message, this delays both the sending and the receiving of the message. Thus, each pilot detects the other pilot is slow and they begin competing causing dependencies which need to be resolved.

Multi-Paxos-Fast-View-Change does not detect the 68 ms slowdown and so the slow leader maintains its leadership. Client commands' end-to-end latency is higher because the 68 ms slowdown is injected for every message: a client command needs to wait 68 ms for the leader to receive the command and 68 ms for each message to commit the entry (one for each follower) plus the normal quorum latency. In the steady-state once it commits and sends another request it will be queued behind the other 6 clients' requests which also take this amount of time. Multi-Paxos-Fast-View-Change quickly detects the 204 ms slowdown and changes leaders.

5.5 Related Work

This section discusses related work on: RSM protocols in general, RSM protocols that provide some form of slowdown tolerance, and slowdown cascades across shards.

RSM protocols. Paxos [35], Viewstamped Replication [46, 57], and Raft [59] are all leader-based replicate state machine (RSM) protocols. These leader-based protocols are, to the best of our knowledge, the state of practice. They are (relatively) simple to implement, provide very good performance, and of course enable RSMs to survive the failure of a minority of replicas. AVICENNA builds on the principles and general structure of these simple, widely-deployed protocols to provide slowdown tolerance.

Mencius uses rotating leaders and noops to optimize for wide-area network deployments [49].

Many other consensus protocols have been proposed that optimize higher throughput, scalability to larger RSMs, and lower latency. Compartmentalization enables the throughput of Paxos to scale up [74]. Fast Paxos decreases latency when concurrent commands do not conflict using larger fast quorums [39]. Generalized Paxos decreases latency when concurrent commands do not conflict or are commutative using fast quorums [37]. Egalitarian Paxos used a leaderless protocol, explicit dependencies, and fast quorums to further optimize for wide-area network deployments [51]. Timestamp ordered queuing is a technique for getting Egalitarian Paxos to take its fast path more often [69]. Domino shows how to use network latency measurements to make Fast Paxos take its fast path more often [75]. Semi-decentralized Paxos overlaps replication and ordering to optimize for wide-area network deployments [77]. All of these protocols have some cases where they provide lower latency in wide-area networks than leader-based protocols like AVICENNA by incorporating more complex mechanisms and often using larger fast quorums. WPaxos allows different replicas to be the leader for different objects and has a lightweight stealing mechanism that lets

the leader easily move around, which optimizes for latency in wide-area networks. An interesting avenue of future work is determining if we can achieve the lower latency of this class of protocols along with slowdown tolerance.

RSM protocols that target some form of slowdown tolerance. Aardvark is designed to be byzantine fault tolerant and prevents a faulty leader from perpetually slowing the RSM [2]. It does this by ensuring a steady stream of PREPREPARE messages and requiring the primary to achieve a continuously increasing throughput threshold. If there is a slowdown from the client to the leader, the client will send the request to every replica, which will eventually initiate a view change when the PREPREPARE for that client request does not arrive. Thus, Aardvark bounds latency during slowdowns (see Table 4.1). However, Aardvarks normal case latency is three wide-area message delays for clients colocated with the leader, and thus is not competitive with the state-of-practice.

Copilot is the first replicated state machine protocol to tolerate a single slowdown [55]. It uses two leaders to propose client commands at the same time and alternate proposals between pilots to prevent conflicting orderings when there are no slowdowns. Copilot without alternating proposals incurs many conflicts and thus requires prohibitively many message delays to commit. Copilot with alternating proposals has high latency in the wide-area because of waiting for the other pilots proposal to arrive before proposing. Latent Copilot has comparable latency to leader-based protocols in the wide-area by only having the latent pilot propose when it detects a slowdown and bounds latency during a slowdown that it can detect [56]. Latent Copilot, however, incorrectly detects certain (non-)slowdowns; for example, when the link between the two pilots is bidirectionally slow each will detect the other as slow consistently hurting performance even if one being persistently acitve is the better choice.

Rabia is leaderless and relies on enough replicas exchanging the same client command [60]. Which client command is exchanged by the replica depends on the timestamp

of the client command. If the same client command is exchanged it can commit in three message delays and so its normal case latency is at least higher than leader-based protocols and Latent Copilot and AVICENNA. Rabia's clients send commands to its proxy replica which forwards the command to the other replicas. If a client proxy is slow than its ε is unbounded in a way similar to Egalitarian Paxos. Consider applying the trivial change of having clients send their commands to all replicas. When exchanged client commands are not the same it takes at least two message delays to be able to propose again. Due to network asynchrony and waiting for a quorum of messages from other replicas, its possible to have different quorums on each replica that contain different client commands with an earliest timestamp. This can result in cases with multiple consecutive instances of conflicting client commands, each requiring three delays to discover and begin the next exchange phase.

Slowdown cascades. Occult shows how taking an optimistic approach can avoid slowdown cascades—where the slowness of a single node cascades and affects other nodes—for a system that provides causal consistency with transactions [50]. Such an approach prevents slowdowns from cascading between shards (replicated nodes). But, it does not address how to prevent slowdowns within a shard (replicated node), which is the focus of AVICENNA. An interesting avenue of future work is unifying the definitions of slowdowns in these two settings and designing a system that can take advantage of both types of techniques.

Chapter 6

Conclusion

This dissertation presented two works that make it easier to build strongly consistent distributed systems that do not get bottlenecked by a single-machine.

The multi-sequence abstraction extends the sequence abstraction to enable consistent ordering across shards with only local ordering information. We proposed the contiguous multi-sequence abstraction for building consistent services. It is a stronger abstraction than the noncontiguous multi-sequence abstraction in use today, making it easier to build services with multi-sequences. We also presented MASON, the first system to expose the contiguous multi-sequence abstraction and the first to provide a scalable multi-sequence. We demonstrated MASON's usefulness as a building block for scalable, consistent services by using it to enable scalability in two services that were previously fundamentally unscalable.

We presented an examination of slowdown tolerance in a wide-area network for the first time and an examination of reactive vs. proactive slowdown tolerant protocols. We presented AVICENNA a replicated state machine protocol that tolerates a slowdown and has latency comparable to the state-of-practice in the wide-area. It uses ghost leaders to explicitly evaluate when another leader would be better and a fast leader-change protocol inspired by early work in conjunction with delegates to rotate quickly.

Bibliography

[1] Remzi Can Aksoy and Manos Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, New York, NY, USA, 2019. ACM.

[2] Lorenzo Alvisi, Allen Clement, Mike Dahlin, Mirco Marchetti, and Edmund Wong. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[3] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: the new SQL server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, New York, NY, USA, 2019. Association for Computing Machinery (ACM).

[4] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Berkeley, CA, USA, 2020. USENIX Association.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, Berkeley, CA, USA, 2012. USENIX Association.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, New York, NY, USA, 2013. Association for Computing Machinery (ACM).

[7] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. Scalable State-Machine Replication. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*, New York, NY, USA, 2014. Association for Computing Machinery (ACM).

[8] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, November 2006.

[9] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'07)*, New York, NY, USA, 2007. Association for Computing Machinery (ACM).

[10] Tushar Deepak Chandra and SAM Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.

[11] Paulo R Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*, Piscataway, NJ, USA, 2017. IEEE.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264. USENIX Association, October 2012.

[13] James A Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference*, volume 12, 2012.

[14] D430. `https://wiki.emulab.net/wiki/d430`, 2021. Accessed: 12-10-2021.

[15] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR'15)*, New York, NY, USA, 2015. Association for Computing Machinery (ACM).

[16] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI'20)*, USA, 2020. USENIX Association.

[17] DPDK Project. DPDK. `https://dpdk.org`, 2021. Accessed: 12-10-2021.

[18] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, New York, NY, USA, 2020. Association for Computing Machinery (ACM).

[19] Robert Escriva, Ayush Dubey, Bernard Wong, and Emin Gün Sirer. Kronos: The design and implementation of an event ordering service. In *Proceedings of the 9th*

European Conference on Computer Systems (EuroSys'14), New York, NY, USA, 2014. Association for Computing Machinery (ACM).

[20] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. White-box atomic multicast. In *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*, Piscataway, NJ, USA, 2019. IEEE.

[21] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.

[22] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[23] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*, New York, NY, USA, 2014. Association for Computing Machinery (ACM).

[24] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[25] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[26] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, Berkeley, CA, USA, 2010. USENIX Association.

[27] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[28] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[29] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11)*, Piscataway, NJ, USA, 2011. IEEE.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*, Berkeley, CA, USA, 2019. USENIX Association.

[32] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Berkeley, CA, USA, 2012. USENIX Association.

[33] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, 2001.

[34] Anthony Kougkas, Hariharan Devarajan, Keith Bateman, Jaime Cernuda, Neeraj Rajesh, and Xian-He Sun. ChronoLog: A Distributed Shared Tiered Log Store with Time-based Data Ordering. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST'20)*, Piscataway, NJ, USA, 2020. IEEE.

[35] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[36] Leslie Lamport. Generalized consensus and Paxos. *Technical Report MSR-TR-2005-33*, 2005.

[37] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[38] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2), 2006.

[39] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[40] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19:104–125, 2006.

[41] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the 28th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'09)*, New York, NY, USA, 2009. Association for Computing Machinery (ACM).

[42] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS'19)*, Piscataway, NJ, USA, 2019. IEEE.

[43] Bojie Li, Gefei Zuo, Wei Bai, and Lintao Zhang. 1Pipe: Scalable Total Order Communication In Data Center Networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[44] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, New York, NY, USA, 2017. Association for Computing Machinery (ACM).

[45] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[46] Barbara Liskov and James Cowling. Viewstamped replication revisited. `http://www.pmg.lcs.mit.edu/papers/vr-revisited.pdf`, 2012.

[47] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, New York, NY, USA, 2011. Association for Computing Machinery (ACM).

[48] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: a partially ordered shared log. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Berkeley, CA, USA, 2018. USENIX Association.

[49] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Berkeley, CA, USA, 2008. USENIX Association.

[50] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[51] Iulian Moraru, David G Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, New York, NY, USA, 2013. Association for Computing Machinery (ACM).

[52] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *Distributed Computing: 13th International Symposium, DISC'99 Bratislava, Slovak Republic September 27–29, 1999 Proceedings 13*, pages 49–63. Springer, 1999.

[53] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[54] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Chariots: A scalable shared log for data management in multi-datacenter cloud environments. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT'15)*, Kostanz, Germany, 2015. OpenProceedings.org.

[55] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Berkeley, CA, USA, 2020. USENIX Association.

[56] Quang Minh Khiem Ngo. *Tolerating Slowdowns in Replicated State Machines*. PhD thesis, Princeton University, 2021.

[57] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A general primary copy. In *ACM Symposium on Principles of Distributed Computing (PODC)*, August 1988.

[58] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, New York, NY, USA, 1988. Association for Computing Machinery (ACM).

[59] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, Berkeley, CA, USA, 2014. USENIX Association.

[60] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying State-Machine Replication Through Randomization. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[61] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4), 1979.

[62] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Berkeley, CA, USA, 2010. USENIX Association.

[63] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Berkeley, CA, USA, 2015. USENIX Association.

[64] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11), 2019.

[65] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Surveys*, 22(4), December 1990.

[66] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, New York, NY, USA, 2001. Association for Computing Machinery (ACM).

[67] The Apache Software Foundation. ZooKeeper Recipes and Solutions. `https://zookeeper.apache.org/doc/current/recipes.html`, 2021. Accessed: 12-10-2021.

[68] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*, New York, NY, USA, 2012. Association for Computing Machinery (ACM).

[69] Sarah Tollman, Seo Jin Park, and John K Ousterhout. Epaxos revisited. In *NSDI*, pages 613–632, 2021.

[70] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Berkeley, CA, USA, 2004. USENIX Association.

[71] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Berkeley, CA, USA, 2017. USENIX Association.

[72] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, Berkeley, CA, USA, 2002. USENIX Association.

[73] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[74] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[75] Xinan Yan, Linguan Yang, and Bernard Wong. Domino: using network measurements to reduce state machine replication latency in wans. In *Proceedings of the 16th*

International Conference on emerging Networking EXperiments and Technologies, pages 351–363, 2020.

[76] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4), 2018.

[77] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building efficient semi-decentralized geo-replicated state machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.

[78] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

Appendix A

Proof of MASON's Strict Serializability

This presents a proof of the strict serializability of assignment of multi-sequence numbers to operations.

A.1 Definitions

Strict serializability requires that there exists a legal total order of operations and that the total order reflects the real-time ordering constraints. Formally: *A complete history h satisfies* linearizability *if there exists a legal total order* τ *of* $ops(h)$ *such that* $\forall op_1, op_2 \in ops(h).op_1 <_h op_2 \Rightarrow op_1 <_\tau op_2$.

That is, if an operation x ends before an operation y begins then x must appear before y in the total order.

Let $\mathscr{S} = \{\mathscr{S}_0, \mathscr{S}_1, ...\}$ be the set of all sequence spaces. We say that two multi-sequence numbers a and b *conflict* if $\exists n \in S$ such that $a_n \neq \Delta \wedge b_n \neq \Delta$. Multi-sequence numbers are ordered by the partial ordering τ over all multi-sequence numbers where $a < b \iff \forall n \in S, a_n \neq \Delta \wedge b_n \neq \Delta \implies a_n < b_n$.

Note that this partial ordering includes the possibility of conflicting multi-sequence numbers not being ordered: i.e. where $a_i < b_i \wedge b_j < a_j$ for some $i, j \in S \implies a \| b$.

Algorithm 1: Sequencer Protocol

```
1  𝒮;                                                  // Set of sequence spaces
2  atMostOnce[];                  // Map of (proxy, seqReqId) to the response
3  activeSequencer ← False;
4  when the sequencer receives a message m, from proxy p do
5      case m = RequestSeqNum(seqReqId, {count_i}_{i=0}^{|𝒮|}) do
6          if ¬activeSequencer then
7              return null;
8          if (p, seqReqId) ∈ atMostOnce then
9              return atMostOnce[(p, seqReqId)], True;
10         resp ← {∅}_{i=0}^{|𝒮|};
11         for i ∈ {0, ..., |𝒮|} do
12             if count_i ≠ 0 then
13                 resp_i ← 𝒮_i;
14                 𝒮_i ← 𝒮_i + count_i;
15         atMostOnce[(proxyId, seqReqId)] ← resp;
16         return resp, False;
17     case m = Recover do
18         for each proxy do
19             send GetMaxAndSeal to each proxy
20         wait for all proxies to reply
           // Portion of recovery for contiguity is omitted.
21         for i ∈ {0, ..., |𝒮|} do
22             𝒮_i ← max_{response∈responses} 𝒮_i in response + 1;
23         activeSequence ← True;
```

The goal of MASON is to provide an ordering for a service built on MASON. Thus, we prove that the partial ordering τ produced by MASON is a legal total order satisfying linearizability. It is then up to the service to apply the operations in the order determined by τ.

An operation is *assigned* a multi-sequence number when the Raft entry containing the operation and multi-sequence number pair is committed. Multi-sequence numbers are *allocated* by the sequencer to a request from the proxy; this does not guarantee the operation

Algorithm 2: Proxy State and Request Protocol

```
1  curSeqReqId ← 0;
2  maxCmtdSeqReqId ← −1;                    // updated in ApplyLog locally
3  cmtdSeqReqIds[];                         // holds all committed SeqReqIds
4  maxRecvdSeqNum[];          // max received sequence number for each
     sequence space
5  sequencers[];                                  // array of sequencers
6  activeIndex ← 0;                         // index of the active sequencer
7  when proxy p receives a message m do
8      case m = ClientRequest(op) do
9          retx ← True;
10         seqReqId ← curSeqReqId;
11         curSeqReqId ← curSeqReqId + 1;
12         activeSequencer ← sequencers[activeIndex];
13         nextSequencer ← sequencers[activeIndex + 1];
14         while retx do
15             send (resp, retx) ← seqnumReq(myProxyId, seqReqId, op.seqReq) to
                activeSequencer;
16             wait for response or suspect activeSequencer has failed;
17             if suspect activeSequencer has failed then
18                 send Recover to nextSequencer
19             wait for response from activeSequencer;
20             if sequencers[activeIndex] ≠ activeSequencer then
21                 return
22             if retx then
23                 executeNoop(resp);
24                 seqReqId ← curSeqReqId;
25                 curSeqReqId ← curSeqReqId + 1;
26         replicate(seqReqId, resp);
27         wait for commit;
28         updateMaxRecvdSeqNum(resp);
29         execute(op);                          // Determined by service.
30         return to op.client;
31     case m = GetMaxAndSeal do
32         activeIndex ← activeIndex + 1;
33         replicate(seal);                        // contains activeIndex
34         wait for commit;
35         return maxRecvdSeqNum;
```

Algorithm 3: Proxy Leader Failover Recovery Protocol

```
1 when proxy replica gains Raft leadership do
2     UncmtdSeqReqIds ← {i.i ∈ ℤ ∧ i ≤ maxCmtdSeqReqId};
3     UncmtdSeqReqIds ← UncmtdSeqReqIds \ cmtdSeqReqIds;
4     for seqReqId ∈ UncmtdSeqReqIds do
5         send (resp, retx) ← seqnumReq(myProxyId, seqReqId, 0);
6         wait for response;
7         if retx then
8             executeNoop(resp);
9     curSeqReqId ← maxCmtdSeqReqId + 1;
```

for which the proxy requested a multi-sequence number will be assigned the allocated multi-sequence number.

We allow operations to be assigned a range of sequence numbers in each sequence space. We will share notation for operations and sequence numbers where for an operation x, x_n denotes the maximum sequence number assigned to operation x in sequence space n. The comparison $x_n < y_n$, and $x_n \leq y_n$ compares the highest assigned sequence number for operation x in sequence space n and the lowest assigned sequence number for operation y in sequence space n That is, $x_n < y_n \iff \max_{i \in x_n} i < \min_{i \in y_n} i$ and $x_n \leq y_n \iff \max_{i \in x_n} i \leq \min_{i \in y_n} i$.

The term *proxy* indicates a replicated state machine that executes the MASON protocol detailed in Alg 2 and Alg 3. *Sequencer* denotes a machine executing the protocol detailed in Alg 1. A standby sequencer may begin executing the sequencer protocol from line 17 of Alg 1 when notified by any proxy.

A.2 Assumptions

The model consists of a set of processes, $\mathscr{P}$, which contains clients, proxy replicas, and sequencers. Processes may fail according to the crash failure model, where processes stop executing requests, and the failure is undetectable to other processes.

We assume an asynchronous network model where messages can be arbitrarily delayed and reordered.

We develop MASON's proxies with Raft and assume the following as guarantees from Raft [59], the guarantee A.2.1 being explicitly stated in the paper.

Guarantee A.2.1. *"If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms."*

Guarantee A.2.2. *Raft is available as long as a majority of replicas have not failed.*

A.3 Proof of total order

To prove that MASON provides a linearizable ordering we first show that its ordering is a total order and then prove that the total order respects the real-time order.

To provide a total order MASON needs to ensure for any two operations x and y one of $x < y$, $y < x$, or $\forall n \in S, x_n = \Delta \vee y_n = \Delta$. The latter case describes when the two operations share no sequence spaces, which we will call *strictly concurrent* and denote $x \|_s y$; in this case x and y are trivially ordered in either order. When any of these relations are true we will say operations x and y are *strictly ordered*. More specifically for any two operations x and y, x and y are strictly ordered if and only if $(x_n < y_n \forall n \in S.(x_n \neq \Delta \wedge y_n \neq \Delta)) \vee (y_n < x_n \forall n \in S.(x_n \neq \Delta \wedge y_n \neq \Delta))$.

We prove that MASON provides a total order, that is, where all operations are strictly ordered as described above.

Lemma A.3.1. *The assigned multi-sequence numbers for any replicated and committed operation do not change.*

Proof: Directly implied by SMR Guarantee A.2.1; any elected leader will have the operation and multi-sequence number pairing in its log. □

The goal then is to prove that any two *assigned* operations are totally ordered, that is we need to show that $\forall x, y, (x < y) \vee (y < x) \vee x \|_s y$. We first prove a total order for conflicting operations.

Lemma A.3.2. *Any two conflicting operations x, y are strictly ordered.*

Proof: We prove by case analysis on all possible combinations of failures of MASON components.

Case 0: No failures. The sequencer trivially guarantees the existence of a total order in normal operation. Consider any two conflicting operations $x, y \in ops(h)$ and any two sequence spaces on which x and y conflict, n, m, such that $x_n \neq \Delta \wedge y_n \neq \Delta \wedge x_m \neq \Delta \wedge y_m \neq \Delta$. Without loss of generality let x arrive at the sequencer before y. Let $\mathscr{S}_n = i, \mathscr{S}_m = j$ when x arrives. Lines 11 – 14 of Alg 1 increment $\mathscr{S}_n$ and $\mathscr{S}_m$ by the respective counts before responding. Once the proxy replicates the assigned multi-sequence numbers for x the assignment does not change by Lemma A.3.1. Then, y, arriving later, must receive $\mathscr{S}_n \geq i + x_n.count, \mathscr{S}_m \geq i + x_m.count$ where $x_n.count$ is the number of sequence numbers requested for $\mathscr{S}_n$ by operation x (line 14 of Alg 1). Thus, $x_n < y_n \wedge x_m < y_m$; that is, they are strictly ordered.

Case 1: Proxy follower failure. This case is equivalent to Case 0 by SMR guarantee A.2.2: proxies execute as normal with a majority of non-failing machines in the proxy.

Case 2: Proxy leader failure. Consider two conflicting operations x, y, and any two sequence spaces on which x and y conflict n, m. Upon proxy leader failure there are four cases.

Case 2a: x and y are assigned (committed) before failure. This case is equivalent to Case 0 by Lemma A.3.1.

Case 2b: Neither x nor y are assigned before failure. When x and y are retransmitted by their clients (not shown) they will be allocated *seqReqIds* greater than *maxCmtdSeqReqId*,

by line 9 of Alg 3 and 10 of Alg 2. Without loss of generality consider x and its *seqReqId*, $x.seqReqId$. If the sequencer has already allocated a multi-sequence number for $x.seqReqId$ the sequencer responds with $retx == True$ and the new leader will allocate a new *seqReqId*, by lines 8 – 9 of Alg 1 and lines 14 – 25 of Alg 2. The $x.seqReqId$ is then incremented and the request to the sequencer is resent lines 14 – 25 of Alg 2. This is repeated, line 14 of Alg 2, until the sequencer has not allocated a multi-sequence number for $x.seqReqId$, indicated by returning $retx == False$, line 16 of Alg 1 and line 14 of Alg 2. x is then allocated a new multi-sequence number. Thus, x and y eventually receive new sequence numbers and this case is equivalent to Case 0.

Case 2c: Either x or y is assigned before failure, and the other is not. Without loss of generality assume x is assigned a multi-sequence number and y is not. The logic is similar to Case 2b. When y is retransmitted by its client (not shown) it will be allocated a *seqReqId* greater than *maxCmtdSeqReqId*, by line 9 of Alg 3 and 10 of Alg 2. Consider y's *seqReqId*, $y.seqReqId$. If the sequencer has already allocated a multi-sequence number for $y.seqReqId$ the sequencer responds with $retx == True$ and the new leader will allocate a new *seqReqId*, by lines 8 – 9 of Alg 1 and lines 14 – 25 of Alg 2. The $y.seqReqId$ is then incremented and the request to the sequencer is resent lines 14 – 25 of Alg 2. This is repeated, line 14 of Alg 2, until the sequencer has not allocated a multi-sequence number for $y.seqReqId$, indicated by returning $retx == False$, line 16 of Alg 1 and line 14 of Alg 2. y is then allocated a new multi-sequence number. Thus, y eventually receives a new sequence number and this case is equivalent to y arriving to the sequencer later as in Case 0. These subcases exhaust all 4 combinations of the state of processing of x and y.

Case 3: Sequencer failure. All multi-sequence numbers replicated (and assigned) before sequencer failure are totally ordered by Case 0 and Lemma A.3.1. What remains to show is that all multi-sequencers assigned after failure are totally ordered. No multi-sequence number allocated by the previous sequencer will be assigned after line 34 of Alg 2 because

of lines 32 and 20 – 21 of Alg 2. Proxies trivially ensure *maxRecvdSeqNum* $\geq$ all assigned multi-sequence numbers at commit time, line 28 of Alg 2. Thus, for any assigned multi-sequence number x at the time of seal commit: $x_i \leq \mathscr{S}_i.i \in \mathscr{S}$, by lines 21 – 22 of Alg 1. For any multi-sequence number, y, assigned after recovery, $\mathscr{S}_i < y_i.i \in \mathscr{S}$, line 14 of Alg 1. So, $x < y$ for any pair (x,y) where x is assigned before recovery seal and y is assigned after recovery seal. $\forall j,k \in ops(h).j,k$ assigned after recovery, j and k are strictly ordered or strictly concurrent by Case 0.

Case 4: Concurrent proxy leader and proxy follower failure. This case is equivalent to Case 2 by the guarantee of availability when fewer than a majority of machines failed A.2.2.

Case 5: Concurrent proxy follower failure and sequencer failure. As a guarantee of SMR, proxies continue to operate as normal with a majority of non-failing machines (A.2.2). Thus, this case is equivalent to Case 3.

Case 6: Concurrent proxy leader and sequencer failure.

Case 6a: The sealing operation on the proxy was not replicated. The new sequencer cannot execute the recovery process until it receives confirmation from every proxy that they were sealed, line 20 of Alg 1. Sealed confirmations are not sent until the seal is replicated. Thus, the new leader will eventually hear, via retransmits, from the new sequencer, and begin replicating the seal, line 33 of Alg 2. Thus, this case is equivalent to Case 3.

Case 6b: The sealing operation on the proxy was replicated. SMR guarantees that only a replica with all committed operations can become the new leader, guarantee A.2.1. Thus, the new leader has the seal operation, and begins to execute recovery. Thus, this case becomes equivalent to Case 3. These two cases are exhaustive as the proxy either committed the seal command or did not at any point in time.

Case 7: Concurrent proxy leader, proxy follower, and sequencer failure. This case is equivalent to Case 6 by the guarantees of SMR when f or fewer replicas fail.

These cases are exhaustive because they are all combinations of possible failures of components in MASON.□

Lemma A.3.3. MASON*'s ordering is a total order, that is,* $\forall$ *assigned operations* $x, y, (x < y) \vee (y < x) \vee x \|_s y$.

Proof: Either x and y conflict or they do not. If x and y conflict, then they are totally ordered by Lemma A.3.2. If they are non-conflicting, then they are strictly concurrent and can be ordered by τ in any order. □

A.4 Proof of real-time order

We need to show for any operation x returned to a client, any operation y invoked after x returned is ordered after x in the total order. We denote the event of the response to a client as $resp(op)$ and the invocation event $inv(op)$.

Lemma A.4.1. *If an operation, x, is assigned a multi-sequence number, n, then a sequencer allocated n for x.*

Proof: Lines 9 – 26 of Alg 2 imply that the proxy only replicates, assigns, an operation if *retx* is False (line 14). This implies the returned n was allocated for x, lines 8 – 16 of Alg 1. □

Lemma A.4.2. *For any two operations x and y,* $resp(x)$ *precedes* $inv(y)$ *in real-time implies* $x < y$.

Proof: Given any two operations x and y and, without loss of generality, assume $resp(x)$ precedes $inv(y)$ in real-time, there are two cases x and y conflict or they do not.

Case 0: x and y do not conflict. In this case x and y are strictly concurrent and can be assigned in either order. We order y after x in the total order.

Case 1: x and y conflict. Given Lemma A.4.1, it is sufficient to show that for any y invoked after $resp(x)$, y is *allocated* a higher multi-sequence number than x, such that $x < y$. There are thus two cases: the sequencer that allocated the assigned multi-sequence number for x allocates the assigned multi-sequence number for y or it does not.

Case 1a: The sequencer that allocated the assigned multi-sequence number for x allocates the assigned multi-sequence number for y. In this case $x < y$ by the normal case ordering. Specifically $resp(x) < inv(y)$ implies that x, being already assigned, arrives to the sequencer before y. Line 11 – 14 of Alg 1 increases all sequence spaces for which x requested a sequence number. Thus, the conflicting sequence spaces are increased. The sequence spaces on any sequencer do not decrease, thus, y is allocated a higher multi-sequence number. So, $\forall n \in \mathscr{S}.x_n \neq \Delta \wedge y_n \neq \Delta, x_n < y_n$, thus $x < y$.

Case 1b: The sequencer that allocated the assigned multi-sequence number for x does not allocate the assigned multi-sequence number for y. Without loss of generality let the sequencer that allocates the multi-sequence number eventually assigned to x be S_x and the sequencer that allocates the multi-sequence number eventually assigned to y be S_y. Because y is assigned a multi-sequence number allocated by S_y and x is assigned a multi-sequence number allocated by S_x and $resp(x) < inv(y)$, S_y must have become the active sequencer after $x.seqnum$ was allocated. To become the active sequencer S_y must have received a *Recover* message from a proxy and executed recovery, receiving the *maxRecvdSeqNum* from every proxy (lines 3, 17, 18 – 20, and 23 of Alg 1 and 31 – 35 of Alg 2). As S_x allocated the sequence number eventually assigned to x, $x.seqnum$ must be *assigned* (replicated) before the proxy receives *GetMaxAndSeal* and replicates the *seal*; otherwise, the proxy would have incremented $activeIndex$ and began to ignore messages from S_x, lines 20 – 21 and 30 – 31 of Alg 2. Thus, $x.seqnum$ is replicated before *seal* and the proxy replies with a multi-sequence

number, *max* such that $\forall n.x_n \neq \Delta, x_n \leq max_n$. Thus, S_y will have $x_i \leq max_i < \mathscr{S}_i \forall i.x_i \neq \Delta$, line 19 of Alg 1 and line 35 of Alg 2. The sequence spaces on any sequencer do not decrease and so $\forall n.x_n \neq \Delta \wedge y_n \neq \Delta, x_n \leq max_n < y_n$. Thus, $x < y$. □

Theorem A.4.1. MASON *provides a strictly serializable total ordering.*

Proof: MASON provides a total order, by A.3.3, that respects real-time ordering, by A.4.2. □