BUILDING A SCALABLE SHARED LOG

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Cong Ding

December 2020

BUILDING A SCALABLE SHARED LOG

Cong Ding, Ph.D.

Cornell University 2020

The *shared log paradigm* is at the heart of modern distributed applications in the growing cloud computing industry. Often, application logs must be stored durably for analytics, regulations, or failure recovery, and their smooth operation depends closely on how the log is implemented.

An ideal implementation of the shared log abstraction should be capable of growing elastically in response to the needs of its client applications, without compromising availability; recover quickly from failures; adopt the data layout that best matches the performance requirement of its clients; scale write throughput without giving up on total order; and offer a low latency that satisfies applications' requirements. Unfortunately, no single shared log today can offer this combination of features. In particular, no shared log provides both total order and *seamless reconfiguration*, i.e., the capability to reconfigure the service without compromising its global availability. Additionally, no shared log provides both total order and low latency.

In this dissertation, we analyze the challenges of achieving both total order and seamless reconfiguration, and of attaining both total order and low latency. Based on this analysis, we have built and evaluated two systems, Scalog and Ziplog, to demonstrate how to address these challenges.

Scalog is a new implementation of the shared log abstraction that offers an unprecedented combination of features for continuous smooth delivery of service: Scalog allows applications to customize data placement, supports recon-

figuration with no loss in availability, and recovers quickly from failures. At the same time, Scalog provides high throughput and total order. At its core is a novel ordering protocol that (1) efficiently merges the order in which records are stored at each shard to produce a single global order across shards, and (2) collates and processes records in the same batch that may originate from any client and be stored at any shard. This novel design enables Scalog to scale to thousands of shards while providing seamless reconfiguration, flexible data placement, and quick failure recovery.

Ziplog is a new implementation of a totally ordered shared log that achieves latency and throughput comparable to what today can only be delivered by systems that optimize only one of these metrics at the expense of the other. Ziplog achieves these results through a new API that, instead of adding new records to the log through a linearizable `Append` operation, relies on a linearizable `InsertAfter` operation that specifies the log position past which the new record should be inserted. This new API allows Ziplog to totally order records across shards without needing cross-shard coordination and with an average latency of fewer than three message delays.

BIOGRAPHICAL SKETCH

Cong Ding spent four wonderful years at Tsinghua Campus and received his Bachelor's degree in Computer Science in 2010 from Tsinghua University. He then moved to the beautiful university town Göttingen and received his Master's degree in 2013 from Georg-August-Universität Göttingen. In 2014, he started his Ph.D. journey at The University of Texas at Austin. In 2016, he moved to Ithaca[1] with his advisor, Professor Lorenzo Alvisi, and continued his Ph.D. study at Cornell University, where Professor Robbert van Renesse joined as a co-advisor.

[1] Ithaca is GORGES!

To my parents and my family

ACKNOWLEDGEMENTS

The journey towards a Ph.D. was wonderful but undeniably long and, at times, tough. I owe a great deal to people who guided and supported me along the way.

First and foremost, I am deeply grateful to my advisors Lorenzo Alvisi and Robbert van Renesse. Lorenzo led me to the magic universe of distributed systems. Robbert opened my eyes to a world I have never seen before, where many beautiful concepts and ideas come. They guided me through the research process and taught me how to present ideas and results. I would like to thank Lorenzo and Robbert for their invaluable advice, endless patience, and continuous encouragement. Their persistence in research inspired me and helped me grow.

I would like to thank other members of my committee for their support and valuable feedback. Li Chen taught me about supply-chain finance, advised me on two projects, and offered suggestions for the dissertation. Rachit Agarwal asked insightful questions, pushing me to think deeper and make this dissertation better.

I am grateful to my collaborators. I have learned a lot from Chunzhi Su, Natacha Crooks, and Chao Xie. Xiang Li, David Chu, Evan Zhao, Chaska Yamane, and Evan Patrick contributed a significant part to work presented in this dissertation. Many other people provided great help and feedback, including but not limited to Sebastian Angel, Mahesh Balakrishnan, Matt Burke, Yaron Minsky, Manos Kapritsos, Youer Pu, Weijia Song, Florian Suri-Payer, Cheng Wang, Yang Wang, and Yunhao Zhang. This dissertation would not have been possible without them.

I would not have finished this journey without the love and support from

my parents. They have unconditionally supported and encouraged me on everything I would love to do.

Finally, I want to thank Subrina, my wonder woman, for being an integral part of my life: believing in me when I doubted myself and cheering me up when I needed it. I am appreciative of her contagious positive nature and remarkable patience with me. My last special thanks go to my daughter, Ivy, who came into my life in the last few months of my Ph.D. journey. With her, the little contribution I make in this dissertation becomes a lot more meaningful.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Shared logs have emerged as a key building block for managing the complexity of building datacenter applications [2, 3, 9, 11, 15, 57]. Their appeal lies in the simplicity of the abstraction they offer: by adding new items to the log's totally ordered sequence of records, clients contribute to building a shared ground truth for their system, which they can then leverage both immediately (e.g., to achieve Paxos-like fault tolerance [26]) and in the background (e.g., to support debugging, analytics, intrusion detection, and failure recovery [12, 47, 57, 82]).

The shared log abstraction provides three basic methods as its API. Clients can *write* records to the log. Once written, records are immutable and must persist. Clients can *read* records from the log. Finally, clients can *subscribe* to the log to be notified when new records are available.

This combination of power and simplicity makes shared logs a popular building block for many modern datacenter applications [2, 3, 9, 11, 15, 57]. All cloud providers offer a shared log service (e.g., AlibabaMQ [2], Amazon Kinesis [3], Google Pub/Sub [9], IBM MQ [11], Microsoft Event Hubs [15], and Oracle Messaging Cloud Service [16]). Shared log services are also available through multiple open-source implementations (e.g., Apache Kafka [57], Corfu [26], and FuzzyLog [68]).

1.1 Shared Log Use Cases

Armed with their simple abstraction and powerful API, shared logs are used by applications in many scenarios.

First, shared logs are used to record and analyze web accesses for recommendations, ad placement, intrusion detection, testing and debugging, etc. [12, 42, 47, 57, 82]. Online applications can record events like page views and user interactions in a shared log. Later, they initiate offline jobs to analyze the log to determine recommendations and ad placement. They can also analyze the log to detect whether the application is compromised and to test and debug the application.

Second, shared logs are used to transfer intermediate results between stages in a processing pipeline that may be replayed for failure recovery [12, 82]. Applications that include processing pipelines typically consist of multiple stages. In one stage, raw input data is consumed from a shared log and then aggregated for further processing in the next stage. When used as a transport, a shared log simplifies the development of each stage, because (1) each stage only needs to read records from the shared log, process the records, and write the results of its processing to the shared log, without cross-stage interaction; and (2) if a stage fails, it can recover by replaying records in the shared log, and reconstruct its previous state without affecting other stages.

More broadly, shared logs are used to address the trade-off between scalability and consistency [83]. Consider, for instance, deterministic databases [27, 53, 54, 78, 79]: retrieving transactions from a single shared log allows these databases to shorten or eliminate distributed commit protocols, avoid distributed dead-

locks, and achieve, in principle, superior transactional scalability [74,78,79].

These different uses of shared logs come together in applications such as online marketplaces, where sellers can list and advertise their merchandise, and buyers can browse and purchase. The marketplace uses a log-based database similar to vCorfu [83] to persist user data, where the shared log simplifies fault tolerance by providing the database with a shared "ground truth" about the system's state. Pageviews and purchases are also stored in a log used for multiple purposes, including extracting statistics (such as the number of unique visitors) and training machine learning algorithms that can recommend merchandise and sellers to buyers.

1.2 Shared Log Requirements

To satisfy the need of the applications described above, a shared log implementation ideally achieves the following properties.

Total order. A total order of records makes it easier for applications to maintain consistency. For example, when developers use the shared log for testing and debugging purposes, they want the log to be totally ordered, so that they can replay the log and reproduce previous results. Similarly, when the application fails, it may want to replay the log to recover its previous state. Additionally, deterministic databases built upon shared logs may rely on the total order to implement strong consistency.

Fault tolerance. The log must be fault-tolerant, as the online services that depend on it should run continuously despite failures. Failure recovery must be

quick to mitigate interruption to online applications.

Scalable throughput. Online services may involve a large number of operations per second and require throughput to scale as needed. Scalability is typically achieved by partitioning the log across multiple shards.

Sharding, in turn, introduces the requirement of **seamless reconfiguration**: when the demands of applications change, the log must reconfigure (adding or removing shards) to adapt; during reconfiguration, the log must remain available. Seamless reconfiguration would allow applications to adapt their capabilities without experiencing hiccups that could negatively affect user experiences in online services, such as real-time online games.

Sharding also introduces the requirement of **flexible data placement**: applications may want to customize data placement among shards to achieve data locality and improve applications' throughput and latency. Flexible data placement would allow applications to adopt the data layout that best matches their performance requirements. Laying out data using an inflexible policy could cause significant performance implications: for example, in log-based databases, reads require playing back a log where relevant updates are interspersed with unrelated records (a potential performance bottleneck), and it is not possible to reduce write latency by writing updates to the closest storage service.

Low latency. Minimizing the time between when a client submits a record and when it learns the record has become persistent is important for applications that are interactive or rely on low latency to reduce the probability of transaction conflicts. Consider, for example, deterministic transactional databases [27,

53, 54, 78, 79] that use a shared log as the storage tier, rely on the shared log to build a total order of transactions, or both. When transactions are interactive, the duration of a transaction is affected by the latency of its reads and writes. High latency in the storage tier causes longer latency for each operation, making transactions take longer. With the same throughput demand from applications, the longer the duration of each transaction, the more likely it is that transactions happen concurrently, causing higher conflict rates. With lock-based concurrency control mechanisms [29], longer transactions thus can result in longer blocking and higher chances for deadlock; with optimistic concurrency control (OCC) [58], they can cause higher abort rates and reduce transactional throughput [49]. For online applications, latency directly affects user experiences perceived by customers, and thus may influence the revenue [21].

1.3 The Challenges

An ideal implementation of the shared log abstraction should be capable of achieving all the above requirements. Unfortunately, no single shared log preceding the work presented in this dissertation can offer this combination of features.

Prior shared log implementations [7,14,26,27] are sharded to achieve scalable throughput, such that load can be distributed among the shards and records can be written to the relatively slow disks in parallel. These implementations offer an append operation to write a record to the end of the log. To obtain both scalability and total order, they decouple ordering from data dissemination. They all use a similar workflow to achieve this: a client first obtains the record's position

in the log via some *sequencer*, and then proceeds to make the record persistent within a shard. This design raises three challenges.

The first challenge is supporting flexible data placement and seamless reconfiguration. The difficulty comes from having to maintain the consistency of the log when failures occur—a dilemma that arises whenever a storage system makes decisions about an item's metadata before making persistent the item itself [25, 46]. Under failure, a record may get lost; this "hole" in the log needs to be filled before other tasks can read beyond the missing entry. Solving this problem requires a costly system-wide agreement on a mapping from log sequence numbers to where records are stored: changes to this mapping are exceedingly expensive, since, until a new mapping is agreed upon, the system cannot operate. For applications using the log, this cost translates into two main limitations. First, they have no practical way of dynamically optimizing the placement of the records they generate, since frequent changes to the mapping would be prohibitively expensive. Second, each time storage servers are added or removed for any reason, they experience a system-wide outage until the new mapping is committed and distributed [26, 27].

The second challenge is that the sequencer can quickly become a bottleneck: designing a sequencer capable of operating at high throughput requires significant engineering effort, frequently involving custom hardware, such as programmable switches (e.g., NOPaxos [67]).

The third challenge is achieving low latency. The difficulty comes from supporting linearizability [50] among a set of operations, including an `Append` operation that adds a new record to the end of the log and other operations to query the log. Prior totally ordered logs use an ordering component, e.g., the

sequencer in Corfu [26], to order records stored in the log and guarantee linearizability for concurrent `Append` operations. To complete an `Append` operation, the shared log must determine where the log ends and insert the record to the tail. However, because the tail might be at any shard, finding the tail requires coordination among all shards. For example, Corfu uses a sequencer to find the tail for clients; when the sequencer fails, each client has to talk to all shards to find the tail. This solution introduces multiple message delays among various servers of the shared log and causes high latency.

1.4 The Systems

We have designed two systems to demonstrate how to address these challenges. Scalog addresses the first two challenges, focusing on seamless reconfiguration, flexible data placement, and scalable throughput in a totally ordered shared log. Ziplog addresses the third challenge, focusing on low latency, without sacrificing other properties.

1.4.1 Scalog

Scalog chooses a new way of decoupling ordering from data dissemination. Instead of assigning to records unique sequence numbers and then performing replication, Scalog takes the opposite: records are first sent to storage servers chosen by clients, replicated, only then assigned a position in the total order.

By adopting a persistence-first architecture, Scalog avoids the first two challenges. It achieves no-downtime reconfiguration, quick failure recovery, and

high throughput, without using custom hardware, via a new, simple, protocol for totally ordering persistent records across multiple shards.

In Scalog, each shard is a group of storage servers that mutually replicate each other's records. Scalable throughput is achieved by creating many high-throughput shards, as in Kafka [82]; however, unlike Kafka, which only provides total order within individual shards, Scalog delivers a single total order across all shards.

Persistence in Scalog is straightforward. A client sends a record to a storage server of its choice, before knowing the record's global sequence number. Storage servers append incoming records, which may come from different clients, to a *log segment* which they replicate by forwarding new records through FIFO channels to all other storage servers within their shard. Thus, each storage server maintains a *primary log segment* as well as backup log segments for every other storage server in its shard. Because of FIFO channels, every backup log segment is a prefix of the primary log segment.

Scalog's second key insight is leveraging the FIFO ordering of records at each storage server to leapfrog the throughput limits of traditional sequencers.

Periodically, each storage server reports the lengths of the log segments it stores to an *ordering layer*. The ordering layer, also periodically, determines which records have been fully replicated and informs the storage servers. Using the globally ordered sequence of reports from the ordering layer, a storage server can interleave its log segments into a global order consistent with the original partial order. Afterward, the storage server can inform clients that their records are both durably replicated and totally ordered.

The ordering layer of Scalog interleaves not only records but also other reconfiguration events. As a result, all storage servers see the same update events in the same order. When a storage server in a shard fails, Scalog's ordering layer will no longer receive reports from the storage server and naturally exclude further records. Other shards are not affected. Thus, clients connected to a shard containing a faulty storage server can quickly reconnect and send requests to servers in other shards.

1.4.2 Ziplog

Besides allowing clients to read the log via the `Read` and `Subscribe` operations, the common shared log API allows them to add a new record R to the log by invoking the `Append`(R) operation, which extends the log by attaching R to its end. To simplify application development, these operations are generally implemented to guarantee *linearizability* [50].

Linearizability helps developers in two critical ways. First, it extends an intuitively sequential notion of correctness to objects that support concurrent operations. For a shared log, linearizability guarantees that all operations issued to the log appear to happen one at a time, and that if operation o_2 starts after o_1 finishes, then o_2 is ordered after o_1. Second, linearizability makes it easier to develop systems modularly by being *composable*: it guarantees that if each object in a system is separately linearizable (i.e, it supports linearizable operations), so is the entire system.

Ziplog's design is motivated by the realization that, in scalable, multi-shard, totally ordered logs, the linearizable append-to-the-end-of-the-log API faces in-

herent high latency costs. The reason is simple: for linearizability, a new append operation must be ordered after any other operation that has completed. Without a separate ordering service, this would require interacting with all shards, which would be hopelessly inefficient. But using a separate ordering service must involve an additional roundtrip message delay. In Corfu [26], which uses a sequencer, this additional roundtrip happens *before* a record is stored, while in Scalog [38], which relies on an ordering layer, it happens *after* a record is stored. Ordering and storing cannot be done concurrently because either the sequence number must be stored with the record (as in Corfu), or the record's position in the storage server must be kept by the ordering layer (as Scalog does through its "cuts"). Further latency is incurred to account for the possibility that the ordering service may fail. In Corfu, if the sequencer fails, each client must contact all shards. To avoid this and the failure recovery latency that it adds, Scalog implements the ordering layer logic using Paxos [61], but at the cost of adding even more message delays to its common case.

Ideally, inserting a record in the shared log would involve only a single shard and no interaction with an ordering subsystem: Ziplog's new API matches that ideal, while ensuring that Ziplog retains the key qualities of today's state-of-the-art shared logs: total order, scalable write throughput, and seamless reconfiguration.

Ziplog abandons the `Append`(R) operation, as it is fundamentally incompatible with low latency; in its place, it offers an API that allows clients to insert a new record *after* another specified record. This API continues to be linearizable: its operations appear to occur sequentially and retain the benefits of composability. However, it offers different semantics than the linearizable `Append`(R)

operation. For example, because all new records are no longer appended at the end of the log, it is not guaranteed that, if an operation to insert R_1 completes before another to insert R_2 starts, R_1 will appear before R_2 in the log. However, if R_2 is causally dependent on R_1, then a client can call `InsertAfter`(R_2, rid_1) to order R_2 after R_1, where rid_1 identifies R_1; if R_1 and R_2 are logically concurrent, insisting on having the log store the first before the second is arguably of limited value in an asynchronous system.

Losing `Append`(R) also means that log positions no longer have to be filled in order: Ziplog's API allows for a position to be filled before those that precede it. Though filling log positions out-of-order is crucial to Ziplog's low latency, it also introduces new technical challenges.

In the absence of failures and reconfigurations, Ziplog's API can be implemented without any global coordination between shards. Each shard maintains a local order among the records that it stores. A stored record R is uniquely identified by a pair $\langle sid, lsn \rangle$, where sid is a shard identifier and lsn an index local to shard sid. Assuming shards are numbered 0 to $N - 1$, it is then possible to assign a global index $gsn = N \times lsn + sid$ to R, extending the local order of each shard into a global one.

If R_1's record identifier rid_1 includes R_1's global index gsn_1, then `InsertAfter(`R_2, rid_1`)` can be implemented by sending R_2 to any shard sid_2, which can then store R_2 and assign it to the next unused local index lsn_2 such that $N \times lsn_2 + sid_2 > gsn_1$. To ensure that the shared log does not contain holes, the shard also assigns *no-op* records to each unused local index smaller than lsn_2.

This ordering scheme requires shards to know N, and thus no longer works

when shards are added or removed—for example, to respond to changing capacity needs or server failures. These events do require some coordination, which Ziplog delegates to a Paxos-based membership management service, but, as we show in Chapter 4, in most cases they cause no hiccups in performance.

1.5 Contributions

In summary, this dissertation makes the following contribution.

- We define the desired properties of shared logs, including scalable throughput, low latency, total order, seamless reconfiguration, flexible data placement, and fault tolerance.
- To achieve seamless reconfiguration and flexible data placement in a shared log, we propose to flip a key design choice of current shared log implementation on its head: whereas they order log entries before replicating them, we replicate records first, and then assign each record a position in the total order.
- To provide a scalable total order, we propose to leverage the FIFO ordering of records at each storage server to leapfrog the throughput limits of traditional ordering components.
- To reduce latency, we propose a new `InsertAfter` interface to add new records to the log. While guaranteeing linearizability, this interface enables individual shards to make consistent decisions about the global order of records they store, avoiding cross-shard coordination in the common case and allowing for a low write latency.

- To reduce latency, we also propose a new replication protocol to make records stable within a shard. Records are ordered in two message delays without contention, and three message delays with contention, allowing for a latency similar to NOPaxos but without the use of special hardware.
- We design and implement Scalog to demonstrate how to achieve seamless reconfiguration, flexible data placement, and scalable throughput in a totally ordered shared log.
  - Scalog provides seamless reconfiguration. Our prototype of Scalog sees no increase in latency or drop in throughput while Scalog is being reconfigured.
  - Scalog offers applications the flexibility to select where the records they produce should be stored. We use this capability to build vScalog, a Scalog-based object store that matches the read latency of vCorfu (and achieves twice its read throughput) while offering stronger fault-tolerance guarantees.
  - Scalog delivers (almost) guilt-free total ordering of log records across multiple shards. While Scalog does not eliminate the trade-off between scalable write throughput and total order, it pushes the pain point much further and scales to thousands of shards.
- We design and implement Ziplog to demonstrate how to achieve low latency without sacrificing other properties.
  - Ziplog's `InsertAfter` interface and new replication protocol allow records to be stable within a shard. Records are ordered in two message delays without contention, and three message delays with contention. Our evaluation shows that contention is rare, allowing for a

latency similar to NOPaxos but without the use of special hardware.

- Ziplog's membership management service, outside of the write critical path, handles only reconfiguration and failure recovery. The light workload of the membership management service makes it able to handle thousands of shards and achieve a throughput comparable to Scalog's.
- Meanwhile, Ziplog provides seamless reconfiguration, without compromising total order or linearizability, again without using special hardware.

1.6 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 provides some necessary background and discusses the design space of shared logs. Chapter 3 presents how Scalog achieves total order, scalability, and seamless reconfiguration. Chapter 4 describes how Ziplog achieves low latency without sacrificing other features Scalog provides. Chapter 5 concludes the dissertation.

CHAPTER 2

BACKGROUND

This chapter discusses how distributed systems achieve each of the shared log properties presented in Chapter 1, illustrates how existing shared log implementations combine these techniques and the trade-offs when implementing each of these combinations, and explores the design space of shared logs.

2.1 General Techniques

2.1.1 Fault Tolerance

Fault tolerance is one of the most important concepts in distributed computing, allowing distributed systems to keep operating properly without losing data or computation, despite some components being faulty. It is generally achieved by state machine replication, which uses multiple replicas: replicas run a consensus protocol (or its equivalent, atomic broadcast) to agree on a sequence of operations; then, each replica individually executes the operations in a deterministic manner—with the same initial state, they produce the same output and achieve the same final state. State machine replication can therefore tolerate the failure of some of its replicas, as the remaining ones will produce the desired output.

Researchers have studied state machine replication for half a century and proposed many consensus protocols and atomic broadcast protocols.

The Primary/Backup approach [31] relies on one primary with one or more

backups. Clients send their operations to the primary, which forwards them to the backups—in this way, all the servers receive the same operations in the same order, and therefore produce the same output and state. Once the primary receives acknowledgments from all backups, it responds to the client that the operation is fully replicated and executed.

Paxos [61,62,80] and its variants [17,45,63–66,69] take a different approach. In Paxos, there are leaders, acceptors, and learners. Clients send operations to at least one of the leaders. Each leader proposes to the acceptors the operations it has received for each log slot. Once a majority of the acceptors have accepted a proposal, the corresponding operation is learned and executed by learners.

Chain Replication [81] organizes replicas into a chain. Clients write to the head of the chain and read from the tail. It achieves high throughput, because each server only talks to its previous server and next server, preventing the primary from becoming a bottleneck as in the Primary/Backup approach.

2.1.2 Scalability

There are two ways to scale a distributed system: scaling horizontally (aka scaling out) and scaling vertically (aka scaling up).

Horizontal scalability is the ability to increase the throughput and capacity by adding more servers. Workloads are partitioned among the servers: each partition deals with a subset of the workload; all partitions work together as a single logical unit. Storage systems, like Cassandra [59], Corfu [26], DynamoDB [37], and Spanner [34], scale to a large number of servers. However,

their throughput and capacity are still limited by their centralized components, such as resource management systems [13] and sequencers [26].

On the other hand, vertical scalability increases throughput and capacity by optimizing software implementations and adding more hardware to a single server. Batching, an example of software optimization, allows servers to process many requests together, which amortizes costs like disk access and network communication. DPDK [6] and SPDK [19] allow the I/O path to bypass the operating system kernel to reduce resource utilization and increase throughput. Hardware solutions include (1) adding more resources such as more memory and additional CPUs, (2) replacing slow hardware with fast ones like SSD, and (3) adopting specialized hardware such as RDMA-based network adaptors.

Horizontal scalability and vertical scalability have their advantages and disadvantages. Achieving horizontal scalability typically needs massive engineering effort, and often, requires redesigning the system. Vertical scalability can be achieved by spending more money on additional hardware. Still, the cost is usually much higher than horizontal scalability and laws of physics limit its throughput and capacity. Real-world systems use a combination of them for their particular needs.

2.1.3 Total Order

An order is a binary relationship that defines which of two distinct operations happens before the other. In a set of totally ordered operations, any pair of operations must have an order, and the orderings are transitive. In other words, operations are totally ordered as a sequence. Ordering client requests is impor-

tant in distributed computing, because applications may rely on the order to deterministically reproduce results, to eliminate deadlocks, or to build a replicated state machine. When the log scales horizontally and is partitioned into a collection of shards, it becomes a challenge to build a total order for records stored in different shards.

DistributedLog adopts a single writer solution. All requests must go through a single server (i.e., the single writer), which orders requests and forwards them to each shard. However, the single write can quickly become a bottleneck and limit throughput.

Corfu [26] and LogDevice [14], instead, use a centralized sequencer to order records, allowing records to bypass the centralized component. These systems first request a sequence number for each record, then process records in the order indicated by sequence numbers. The sequencer can handle more throughput than the single writer in DistributedLog, but it still limits the maximum throughput.

Mencius [70], Calvin [79], and Derecho [52] use the round-robin approach and aim for scale by having clients send requests directly to individual shards. All shards, ordered by their shard identifiers, take turns to order their receiving requests. This round-robin approach, in theory, scales arbitrarily, but it drastically increases overall latency, especially when the number of shards is large and at least one of the shards is slower than others.

2.1.4 Seamless Reconfiguration

Vertical Paxos [65] supports seamless reconfiguration as part of agreeing on the sequence of operations—however, it does not support sharding and therefore, cannot scale.

Horizontally scalable systems desire to add and remove shards on the fly, while vertically scalable systems may want to replace servers without stopping the service. The ability to reconfigure is essential, but it is hard to achieve it seamlessly: most systems suffer from "hiccups" during reconfiguration.

Systems relying on ZooKeeper [51] to reconfigure typically temporarily pause during reconfiguration. For example, Corfu [26] takes about 30 ms to complete a reconfiguration.

2.1.5 Flexible Data Placement

Flexible data placement typically implies that clients select the shard to store their data. By default, sharded systems should support flexible data placement; however, supporting other properties often limits flexible data placement. For example, Corfu [26] relies on an inflexible data placement policy to expedite failure recovery: when a client fails, another client can easily find out the shard that stores partial data from the failed client and complete the partially written data.

2.1.6 Low Latency

The time that elapses from when a client issues a request to when it receives a response is an important metric. As discussed in Chapter 1, low latency may increase throughput and improve user experience. Distributed systems builders have put much effort into reducing latency.

Zyzzyva [55], EPaxos [71], Speculative Paxos [73], and NOPaxos [67] speculate on different aspects of a system's behavior, including the correctness of the leader, dependencies among messages, and message order in the network. When speculation is successful, the systems proceed fast; otherwise, they must fix the speculated results, taking additional time.

Mencius [70] allows clients to talk to local servers, where the local server acts as a leader for a subset of the Paxos instances. This approach avoids expensive cross-datacenter traffic and reduces latency.

RDMA, DPDK, and SPDK are techniques that allow data to bypass the kernel, which enables systems not only to scale vertically (see Section 2.1.2), but also to reduce latency. Derecho [52], a persistent group communication facility, relies on an RDMA infrastructure to reduce latency. Arrakis [72] and IX [28] implement library operating systems that allow the I/O path to bypass the kernel, which, similar to DPDK and SPDK, avoids the cost of system calls and reduces latency.

2.2 Prior Shared Log Implementations

2.2.1 Totally Ordered Shared Log

As discussed in Chapter 1, the total order property of shared logs makes it easier for applications to maintain consistency. Therefore, researchers design totally ordered shared logs by relaxing other properties.

Corfu

To provide both total order and high throughput, Corfu [26] separates ordering from data dissemination and relies on sharding. A function, maintained in ZooKeeper [51], maps sequence numbers to shards. A client first obtains a sequence number for a record from the Corfu sequencer, and then forwards the record to the shard indicated by the mapping function. Each shard comprises a collection of replicas, each consisting of a *flash unit* (an SSD plus FPGA to implement a write-once block device), kept consistent using a variant of chain replication [81].

Corfu can adapt to applications' needs by adding or removing storage servers while maintaining total order across records stored on different servers. However, any change in the set of storage servers makes Corfu unavailable until the new configuration is committed at all storage servers and clients.

Second, the Corfu data layout is defined by an inflexible round-robin policy. As discussed in Section 1.3, the inflexible data placement policy can cause significant performance implications: it prevents applications from adopting the

data layout that best matches their performance requirements.

Third, the tension between scaling across multiple storage servers and guaranteeing total order ultimately limits Corfu's write throughput. The optimized Corfu implementation used in Tango [27] achieves, to the best of our knowledge, the best throughput among totally ordered shared logs prior to this dissertation; but, at about 570K writes/sec, its performance still falls short of the needs of the most demanding applications. For example, Taobao [20], Alibaba's online market, ran millions of database writes/sec at its 2017 peak [1].

Finally, the latency figures of Corfu can have serious negative implications for some of the applications that rely on them. Adding a new record to Corfu takes over 1 ms. NOPaxos [67], in comparison, can complete consensus (and thus append a record to a shared log) in 111 μs to 200 μs, depending on whether special hardware is used—but only on a single shard. It is desirable to build a totally ordered shared log, with throughput similar to Corfu's and latency comparable to NOPaxos's.

vCorfu

vCorfu [83] is an object store based on Corfu. Object stores require data locality for fast read; however, vCorfu's underlying storage, Corfu, offers an inflexible data placement policy, which becomes an obstacle for achieving data locality. Instead, vCorfu complements the Corfu shared log with *materialized streams* and employs log-like data structures that store together updates that refer to the same object for flexible data placement and fast read. For this gain in performance, vCorfu pays in robustness: whenever a log replica and a stream replica

fail concurrently—a more likely event as the system scales up [33]—vCorfu is at risk of losing data.

LogDevice

LogDevice [14] is similar to Corfu, but replaces the mapping function with a non-deterministic record placement strategy, which allows for a flexible data placement policy and avoids the problem caused by Corfu's inflexible round-robin policy. However, the price that LogDevice pays is slow failure recovery: when a client fails with a partially completed write operation, another client must contact all the shards to find out where the partial write operation happened and help to complete this operation. Additionally, all records still need to be ordered by a sequencer, limiting throughput, causing high latency, and sacrificing seamless reconfiguration.

2.2.2 Locally Ordered Shared Log

As discussed above, existing totally ordered shared logs, including Corfu, vCorfu, and LogDevice, cannot achieve all the properties shared logs desire; therefore, researchers and software engineers have proposed to trade total order for other shared log properties and have built *locally ordered* shared logs.

Locally ordered shared logs partition the log into multiple shards. Within a shard, records are ordered and replicated using one of the distributed replication protocols. However, records stored in different shards are not ordered with respect to one another.

Kafka

Kafka [57], deployed widely in industry, partitions the log into a collection of shards to scale out and uses a Primary/Backup protocol [31] to replicate records stored within a shard, without providing total order across shards. Kafka's simple design allows it to easily offer all the properties discussed in Chapter 1, except for total order: Clients can write to any of the shards; thus, Kafka supports flexible data placement. When shards are added or removed, no other shards need to be coordinated with; therefore, the reconfiguration is seamless. This design allows Kafka to scale, in theory, to an arbitrary number of shards. However, the maximum scale is limited by other factors, including the network, the size of a datacenter, and the scale a resource management system can handle.

As discussed in Chapter 1, many applications require the shared log to have a total order. Lacking total order, Kafka can only support a limited scope of applications [56]; often, application developers have to play the challenging game of deciding how to partition data, to match what applications need [40].

Derecho

Derecho [52] is a sharded and persistent group communication facility, built over an RDMA infrastructure, aiming to reduce latency. The best-case end-to-end latency is four message delays, which is the same as that of Kafka. However, leveraging RDMA's ultra-low latency (1.5 μs vs. a minimum of 36 μs for TCP/IP), end-to-end latency in Derecho for two replicas is only 50 μs.

2.2.3 Partially Ordered Shared Log

Given that locally ordered shared logs limit the scope of applications and are difficult to use, some researchers have explored a different design point: they have built *partially ordered* shared logs, which performs similar to locally ordered shared logs.

FuzzyLog

FuzzyLog [68] implements a partially ordered shared log. It uses chain replication [81] to order and replicate records within a shard. A variant of Skeen's algorithm [48] tracks Lamport's happened-before relation [60] between records stored in different shards and builds dependency relationships (i.e., a partial order) for these records when the network is not partitioned; when the network is partitioned, FuzzyLog does not build the partial order. FuzzyLog offers "best-effort" partial order: it partially orders records stored in different shards only when possible.

When successful, FuzzyLog can build a partial order at nearly no cost and offers flexible data placement, seamless reconfiguration, and scalable throughput. FuzzyLog's ordering guarantee is strictly stronger than Kafka's, but still weaker than a total order.

2.3 Designing an Ideal Shared Log

Current shared log implementations achieve some of the shared log properties and give up others because of the challenges to implementing an ideal shared log. In this chapter, we present the design space of shared logs and show how to build an ideal shared log from ten thousand feet.

We recognize that not all applications require a global total order, and many industrial applications have been built using shared logs such as Kafka [57] that only provide a total order per shard. However, our unique way of providing total order comes at practically no cost to throughput even under reconfiguration, while latency is no more than that of non-sharded replication protocols and non-totally ordered logs. Thus, programmers need only consider performance when deciding how to shard their applications, and not engage in the difficult balancing game between achieving the required throughput and correctness [23, 40, 41, 74]. Also, a global total order supports reproducibility, simplifying finding bugs in today's complex distributed applications, as discussed in Chapter 1.

2.3.1 Failure Model and Assumptions

Accurate failure detection is impossible [32, 44], so a shared log implementation should assume a crash failure model with asynchronous communication channels: servers may crash or stop to respond for an indefinitely long time, and message delivery time has no upper bound.

Indeed, this failure model is weaker (and hence leads to inherently more

robust systems) than the model used by prior totally-ordered shared logs [26, 27, 83]: it assumes that faulty servers will *crash* [43], rather than *fail-stop* [76], thus sidestepping the so-called "split-brain syndrome" [36].[1]

2.3.2 The Design Space of Totally Ordered Shared Logs

Common shared log implementations decouple ordering from replication for scalable throughput. Different combinations of ordering techniques and replication protocols render different designs with various trade-offs. This section discusses the ramifications of various combinations used in Corfu and Scalog, as well as the semi-decentralized ordering mechanism used in Ziplog.

To replicate records, shared log designs can select from various existing replication protocols, including Paxos [62], Primary/Backup [31], and Chain Replication [81]. They can also resemble Primary/Backup for lower latency: a client sends a record to the primary and all backups and receives responses from each of them; the client can then determine whether all the servers have stored the record in the same log slots—if they have not, the primary will eventually synchronize its decision to the backups. Shared logs can select one of the replication protocols based on the need for their design and applications. For example, Corfu uses client-driven Chain Replication in combination with FPGA-based SSDs for energy efficiency: a client sends a request to the head of the chain, waits for a response, then sends the request to its successor, and so on until it receives a response from the tail on the chain. In contrast, the normal communication pattern in Scalog and Ziplog resembles that of Primary/Backup.

[1] The essential difference is that crash failures cannot be accurately detected, while in the fail-stop model, it is assumed that failures can be detected accurately by some oracle. Violation of this assumption can lead to inconsistencies.

There are also various options to order records. Corfu uses an (unreplicated) sequencer to assign a sequence number to each record before performing replication. If the sequencer fails, clients must instead communicate with all shards and compete for a sequence number for each record, leading to low throughput. If a record is ordered but lost before it is replicated, clients must fill the slot with a no-op before normal operation can continue, causing unavailability.

Scalog uses a replicated "ordering layer" to order replicated records in batches. While Scalog does not lose availability during failure recovery, write latency suffers from batching, and the message delays caused by the ordering layer.

Ziplog's approach allows each shard to individually assign sequence numbers to records and replicate them while relying on a membership management component to allocate sequence numbers for future records. Because the ordering decision for a record is made within a shard, write latency is minimized. However, for this to work well, Ziplog must accurately predict when records are going to arrive at each shard—if mistaken, the sequential access latency could grow significantly. Fortunately, as we will see in Section 4.4.4, the sequential access latency does not grow indefinitely in datacenter networks, and the tail latency is, in fact, lower than that of Scalog.

All the shared logs acknowledge to clients the completion of write operations after both ordering and replication are completed and guarantee linearizability. In Corfu, it is possible to do early acknowledgment: after a record is ordered, but before it is replicated. The early acknowledgment allows clients to perform more aggressively, but at risk of violating linearizability when failure happens. Similarly, Scalog could acknowledge after a record is replicated, but before it is ordered, but the risk is that the record may never be ordered

when failure happens. Ziplog, which performs replication and ordering logically atomic, cannot do early acknowledgment, but its latency is similar to that of Corfu's early acknowledgment and lower than that of Scalog's early acknowledgment.

CHAPTER 3

SCALOG: SEAMLESS RECONFIGURATION AND TOTAL ORDER IN A SCALABLE SHARED LOG

This chapter presents the design, implementation, and evaluation of Scalog, a shared log that simultaneously achieves total order, scalability, and seamless reconfiguration.

Scalog, the new shared log that this chapter introduces, aims to address the limitations in existing systems, as discussed in Chapter 1. Like Corfu, Scalog can scale horizontally by adding *shards* and guarantees a single total order for all records, across all shards. However, reconfiguring Scalog by adding or removing shards requires no global coordination and does not affect its availability. Further, Scalog's API gives applications the flexibility to select which records will be stored in which shard: this allows Scalog to replicate the functionality offered by vCorfu's materialized streams without trading off robustness. Indeed, Scalog operates under weaker failure assumptions (and hence is inherently more robust) than prior totally-ordered shared logs [26, 27, 83]: it assumes that faulty servers will *crash* [43], rather than *fail-stop* [76], thus sidestepping the so-called "split-brain syndrome" [36].[1] Finally, though Scalog cannot scale write throughput indefinitely, it can deliver throughput almost two orders of magnitude higher than Corfu's, with comparable latency.

Scalog's properties derive from a new way of decoupling global ordering from data dissemination. Decoupling these two steps is not a new idea. For example, in Corfu, the sequencer that globally orders records is not responsi-

[1] The essential difference is that crash failures cannot be accurately detected, while in the fail-stop model, it is assumed that failures can be detected accurately by some oracle. Violation of this assumption can lead to inconsistencies.

ble for their replication; once the order has been decided, replication is left to the clients, thus allowing data dissemination to scale until ordering ultimately becomes the bottleneck.

The key to Scalog's singular combination of features is to turn on its head how decoupling has traditionally been achieved. In Corfu (as well as Facebook's LogDevice [14]), order comes before persistence: records are first assigned unique sequence numbers in the total order, and then replicated; in Scalog, the opposite is true: records are first replicated, and only then assigned a position in the total order.

As mentioned above, Corfu requires that all clients and storage servers hold the same function to map sequence numbers to specific shards, causing Corfu to be temporarily unavailable when shards are added or removed. By ordering only records that have already been replicated, Scalog sidesteps the need to resolve the delicate case in which a client, having reserved a slot in the total order for one of its records, fails before making that record persistent. Without this burden, Scalog can seamlessly reconfigure without any loss of availability, and give applications the flexibility to independently specify which shards should store their records, thus matching vCorfu's data locality without the need of dedicated stream replicas.

Specifically, Scalog clients write records directly to storage servers, where they are (trivially) FIFO ordered without the mediation of a global sequencer. Records received by a storage server are then immediately replicated across the other storage servers in the same shard. To produce a total order, Scalog periodically interleaves the FIFO ordered sequences of records stored at each server.

We recognize that not all applications require a global total order, and many industrial applications have been built using shared logs such as Kafka [57] that only provide a total order per shard. However, our unique way of providing total order comes at practically no cost to throughput even under reconfiguration, while latency within a datacenter is no more than a few milliseconds. Programmers thus only need to consider performance when deciding how to shard their applications, an otherwise difficult balancing game between achieving the required throughput and correctness [23,40,41,74]. Also, a global total order supports reproducibility, simplifying finding bugs in today's complex distributed applications.

Our evaluation of a Scalog prototype implemented on a CloudLab cluster [4] confirms that Scalog's persistence-first approach comes closer to an ideal implementation of the shared log abstraction in three main respects:

3.1 Scalog's API

Scalog provides the abstraction of a totally ordered shared log. Table 3.1 presents a simplified API that omits support for authentication and authorization, as well as the ability to subscribe only to records that satisfy a specific predicate.

The first three methods are sufficient for most applications. The `append` method adds a record to the log. When it returns, the client is guaranteed that the record is *committed*, meaning that it cannot be lost (it has been replicated onto multiple disks) and that it has been assigned a *global sequence number* (its unique log position among committed records). The `trim` method allows a prefix of the log to be garbage collected. Finally, the `subscribe` method subscribes to com-

`append`(r)	Append record r, and return the global sequence number.
`trim`(l)	Delete records before global sequence number l.
`subscribe`(l)	Subscribe to records starting from global sequence number l.
`setShardPolicy`(p)	Set the policy for which records get placed at which storage servers in which shards.
`appendToShard`(r)	Append record r, and return the global sequence number and shard identifier.
`readRecord`(l, s)	Request the record with sequence number l from shard s.

Table 3.1: Scalog API

mitted log records starting from global sequence number l. Scalog guarantees that (1) if `append`(r) returns a sequence number, each subscriber will eventually deliver r; and (2) any two subscribers deliver the same records in the same order.[2] Note that these guarantees are sufficient to implement a replicated state machine [77] using Scalog.

To achieve high throughput and support flexible allocation of resources, Scalog structures a log as a collection of *shards*, each in turn containing a collection of records. Applications can exploit the existence of shards to optimize performance with the remaining three API methods. The `setShardPolicy` method lets applications specify a function used to assign records to storage servers and shards. The `appendToShard` method behaves as `append`, but in addition returns the identifier of the shard where the record is stored. The `readRecord` method allows random access to records by sequence number, assuming the shard identifier is known (e.g., for having been returned by a prior invocation of `appendToShard`). Under concurrent access, Scalog offers fully linearizable semantics [50]—the strongest possible consistency guarantee.

[2]These guarantees hold only in the absence of trimming. Trimmed records may never be delivered to some subscribers.

Besides this API, Scalog provides various management interfaces that allow reconfiguring the log seamlessly in response to failures and to the changing needs of the applications it serves. Specifically, Scalog can create new shards on-the-fly (if load increases), as well as turn shards from *live* to *finalized*. New records can only be appended to live shards; once finalized, a shard is immutable.

Finalizing shards serves three purposes. First, Scalog optimizes finalized shards for read throughput. To prevent read-heavy analytics workloads from affecting the performance of online services, an operator may create new shards, finalize the old shards (effectively, creating a checkpoint), and then run the analytics workload on the finalized shards. Second, when a storage server in a shard fails, append throughput may be affected; rather than recovering the failed server, Scalog allows finalizing the entire shard and replacing it with a new one (see Section 3.2). If one wishes to restore the level of durability, additional replicas may be created after a shard is finalized. Third, finalized shards may be garbage collected: this is how resources are reclaimed after a log is trimmed.

3.2 Scalog's Architecture

Figure 3.1 presents an overview of Scalog's architecture, highlighting its three components: a *client library*, used to issue `append`, `subscribe`, and `trim` operations; a *data layer*, consisting of a collection of shards, storing and replicating records received from clients; and an *ordering layer*, responsible for totally ordering records across shards.

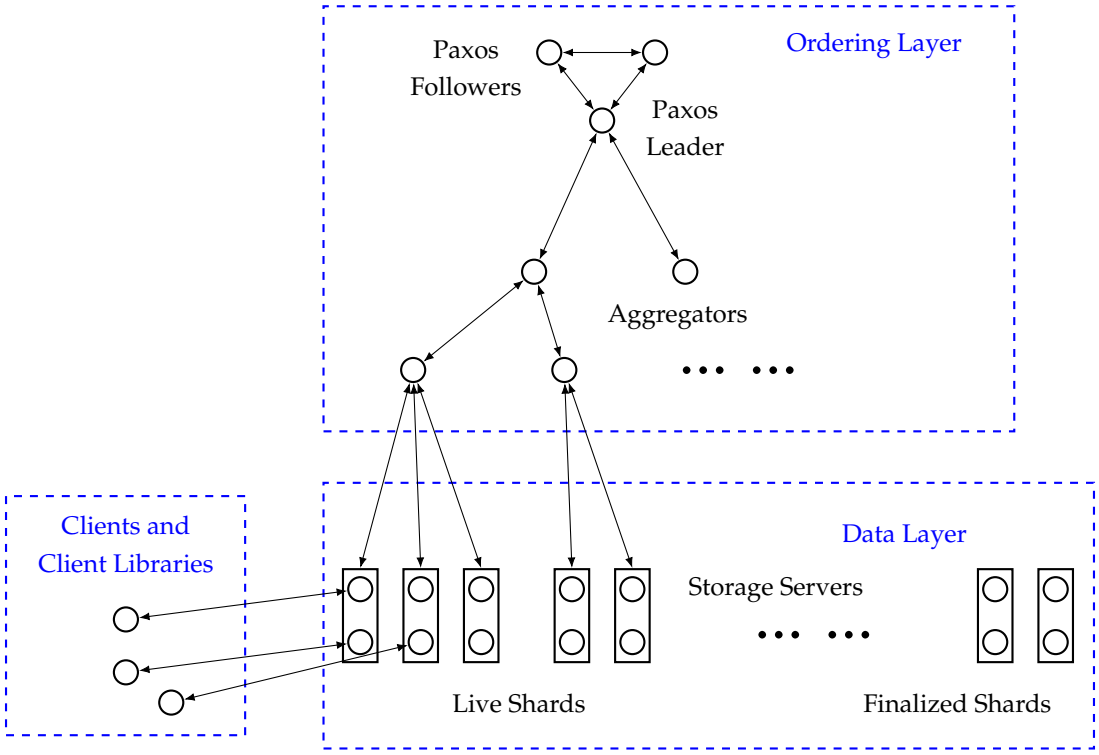


Figure 3.1: Scalog's architecture: arrows denote communication links; circles denote servers; each rectangle denotes one shard. Servers in the same shard communicate with each other. In this example, both shards and the Paxos instance in the ordering layer are configured to tolerate one crash.

Client Library. The library implements the Scalog API and communicates with the data layer (see Table 3.1).

Data Layer. Scalog's data layer distributes load along two dimensions: each log consists of multiple shards, and each shard consists of multiple storage servers. Each storage server is in charge of a *log segment*. Clients send records directly to a storage server within a shard. When a storage server receives a record from a client, it stores the record in its own log segment. For durability, each server replicates the records in its log segment onto the other storage servers in its shard. To tolerate f failures, a shard must contain at least $n = f + 1$ storage servers.

Ordering Layer. The ordering layer periodically summarizes the fully replicated prefix of the primary log segment of each storage server in a *cut*, which it then shares with all storage servers. In a Scalog deployment with m shards, each comprising n storage servers, the cut has $m \cdot n$ entries, each mapping a storage server identifier to the sequence number of the latest durable record in its log segment. The storage servers use these cuts to deterministically assign a unique global sequence number to each durable record in their log segments. Besides enabling global ordering, the ordering layer is also responsible for notifying storage servers of reconfigurations.

The ordering layer must address two concerns: fault tolerance and scalability under high ordering load. Scalog addresses the first concern by implementing the ordering layer logic using Paxos [61]. The second concern is that the overhead of managing TCP connections and handling the ordering requests could overwhelm the ordering layer when there are a large number of shards. This concern is addressed with the help of the aggregators, illustrated in Figure 3.1. Scalog spreads the load using a tree of aggregators that relay ordering information from the storage servers at the leaves up to the replicated ordering service. Each leaf aggregator collects information from a subset of storage servers (we assume servers in the same shard report to the same leaf aggregator) and determines the most recent durable record in their log segment before passing the information up. Aggregators use soft states and do not need to be replicated—if suspected of failure, they can easily be replaced. The ordering reports passed up the tree are self-sufficient, and need not be delivered in FIFO order. The aggregator tree is maintained by the replicated service in its root—no decentralized algorithms are needed to eliminate loops and orphans.

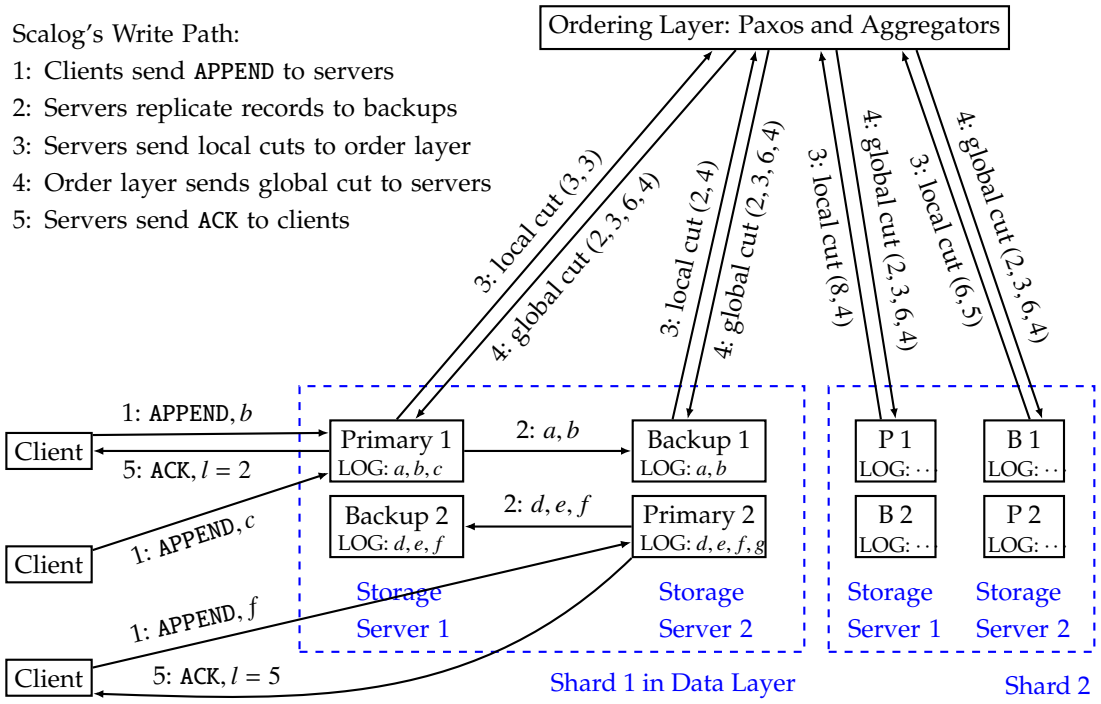


Figure 3.2: Scalog message flow for append operations

3.3 Scalog's Workflow

To further elucidate how Scalog works, we present a detailed explanation of the execution paths for *append*, *read*, and *trim* (garbage collection) operations.

3.3.1 Append Operations

When an application process first invokes its client library to start appending data to the log, the client library chooses a shard according to the current sharding policy set by `setShardPolicy`(p). If no policy has been specified, Scalog applies its default selection policy, choosing a random storage server in a random live shard as the write target.

Having established a destination shard s and storage server d, the application process can add records to the log. When `append`(r) or `appendToShard`(r) is invoked, the client library forwards record r to storage server d in an `APPEND` message (Figure 3.2, Step 1) and awaits an acknowledgment.

As shown in Step 2 of Figure 3.2, each storage server replicates in FIFO order the records it receives onto its peer storage servers in s. In-shard replication resembles Primary-Backup (PB) [24, 31]: each storage server acts as both Primary for the records received directly from clients and Backup for the records in the log segments of its peer storage servers in the shard. Scalog differs from PB in how storage servers learn which records in their log segment have become durable. Instead of relying on direct acknowledgments from its peers, each storage server periodically reports to the ordering layer a *local cut*—an integer vector summarizing the records stored in this storage server's log segments (Step 3 in Figure 3.2). Because log segment replication occurs in FIFO order, each integer in the local cut is an accurate count of the number of records stored in the corresponding log segment.

The ordering layer combines these local cuts to determine the latest durable record in the log segment of each storage server. Let v_i be the local cut for server i in shard s; $v_i[i]$ represents the number of records in i's log segment (i.e., those that i, serving as Primary, received directly from its clients) while $v_i[j], j \neq i$, is the number of records that server i is backing up for its peer storage server j. The ordering layer can then compute the number of durable records in i's log segment as the element-wise minimum of all $v_j[i]$ for all storage servers j in s. For instance, assume $f = 1$ and suppose the ordering layer has received from the two storage servers r_1 and r_2 in shard s the local cuts $v_1 = \langle 3, 3 \rangle$ and

$v_2 = \langle 2, 4 \rangle$. Then, $\langle 2, 3 \rangle$ expresses all durable records in shard s (see Shard 1 in Figure 3.2). By repeating this process for all storage servers in every shard, the ordering layer assembles a *global cut*, a map that represents records stored in all log segment replicas and therefore durable. To prevent the number of entries in global cuts from growing indefinitely, we use a single integer to represent the total number of records in all finalized shards. The ordering layer then forwards each global cut to all storage servers (Step 4 in Figure 3.2).

The totally ordered sequence of cuts can be used to induce a total ordering on individual records. Summing the sequence numbers in the elements of a cut gives the total number of records that are ordered up to and including that cut. The difference between any two cuts determines which records are covered by those two cuts. We use a deterministic function that specifies how to order the records in between two consecutive cuts. In our current implementation, we use a simple lexicographic ordering: records in lower-numbered shards go before records in higher-numbered shards, and within a shard records from lower-numbered storage servers go before records from higher-numbered storage servers.

Therefore, upon receipt of a global cut, a storage server can determine which records in its primary log segment are now globally ordered, and then acknowledge the corresponding append requests by returning the record's global sequence number to the client (Step 5 in Figure 3.2). Should a storage server fail, a client can ask any of its backup for the current status of its records.

Note that the load on the ordering layer is independent of the write throughput—it only depends on the number of storage servers and the frequency of their reports.

3.3.2 Read Operations

Applications can read from the log either by subscribing or by requesting specific records. The `subscribe` operation broadcasts the request to a random storage server in each shard. Upon receiving a `subscribe`(l) request, the storage server sends the client all records it already stores whose global sequence number is at least l and then continues forwarding future committed records.

Recall that an application process, by calling `appendToShard`(r), obtains the shard identifier s that stores record r, as well as its global sequence number l. If at a later time the process needs to bring r back in memory, it can do so by invoking `readRecord`(l, s). In response, the client library contacts a random storage server in s to read the record associated with global sequence number l. The receiving storage server then computes $l_{\max}$, the largest global sequence number it has observed, by applying the deterministic scheme of Section 3.3.1 to the latest cut received from the ordering layer, and proceeds to compare l and $l_{\max}$. If $l \leq l_{\max}$, the storage server uses l to look for r in its local log and, if it finds it, returns it; otherwise, if the record has been trimmed (see Section 3.3.3), it returns an error. If $l > l_{\max}$, the storage server waits for new cuts from the ordering layer and updates $l_{\max}$ until $l \leq l_{\max}$; only then does it proceed, as in the previous case, returning to the application process either r or an error message. Allowing responses only from storage servers for which $l \leq l_{\max}$ is critical to guarantee linearizability for concurrent read and append operations, as it prevents stale storage servers from incorrectly returning error messages. The client library may, however, timeout, waiting for a storage server to respond; if so, the client library contacts another storage server in s (the storage server holding r in its log segment is guaranteed to eventually respond).

3.3.3 Trim Operations

Calling `trim`(l) garbage collects the log prior to the record with global sequence number l. The client library broadcasts the `trim`(l) operation to all storage servers in all shards; upon receipt, they proceed to delete the appropriate prefix of the log stored in their respective log segments.

3.3.4 Reconfiguration and Failure Handling

Reconfiguration can happen often in Scalog, not only to recover from failures (which are more likely as scale increases), but also to handle growing throughput or needed capacity. For example, an application that needs to run a read-intensive analytics job can finalize the shards storing the relevant data, making them read-only. For these reasons, Scalog strives to make adding and finalizing shards seamless.

Adding and Finalizing Shards

Adding a new shard is straightforward: as soon as the shard and its servers register with the ordering layer, the new shard can be advertised to clients. Other shards are unaffected, but for the larger-sized cut, its storage servers will henceforth receive from the ordering layer.

We distinguish two types of shard finalization: scheduled finalization and emergency finalization. Scheduled finalizations are initiated in anticipation of shard workload changes. To transition clients off of shards facing impending

finalization, Scalog supports a management operation that causes the ordering layer to stop accepting ordering reports from a shard after a configurable number of committed cuts. This gives clients a "grace period" so that they can smoothly transition to another live shard. Emergency finalizations are needed when a server in a shard fails (see *Finalize & Add* in Section 3.3.4); these failed shards are finalized immediately.

Handling Storage Server Failures

Failing or straggling storage servers are detected either by Paxos servers directly connected to them or by aggregators. Problems are notified to the ordering layer, which in turn initiates reconfiguration. Applications have three options to configure how Scalog handles slow or failed storage servers.

Finalize & Add (Requires at least $f + 1$ storage servers per shard): If a storage server is suspected of having failed or is intolerably slow, its entire shard s is finalized. Clients of s can redirect their writes to other shards; concurrently, a new shard is added to restore the log's overall throughput. Because the ordering layer totally orders finalization operations and cuts, the latest cut before s is finalized reveals which records s successfully received and ordered: these records can be retrieved from any of the surviving storage servers in the shard. Records received but not incorporated in s's latest cut must be retransmitted by the originating clients to different shards. Corfu also responds to a storage server failure by finalizing its shard and adding a new one. During this process, however, all Corfu's shards are unavailable; in contrast, Scalog's non-faulty shards are unaffected (see Section 3.5.2).

	Replicas per Shard	Data Locality	Service Recovery Time
Finalize & Add	$f + 1$	No	Short
Remove & Replace	$f + 1$	Yes	Long
Mask	$2f + 1$	Yes	Zero
Corfu	$f + 1$	No	Short
vCorfu	$f + 1$	Yes	Long

Table 3.2: Trade-offs of different approaches to handling storage server failures

Applications that require data locality may run application processes in storage servers (see Section 3.4.3). *Finalize & Add* would force those processes, if an entire shard is finalized, to migrate. Instead, Scalog supports two alternative options.

Remove & Replace (Requires at least $f + 1$ storage servers per shard): As in vCorfu, Scalog can replace a failed storage server with a new one, which can then copy records from its shard's surviving storage servers. During this process, the affected shard is temporarily unavailable for writes (but continues to serve reads). This option suffers from a longer service recovery time [33].

Mask (Requires at least $2f + 1$ storage servers per shard): At the cost of extra resources, this option ensures that, if no more than f of its storage servers fail, a shard will continue to process both reads and writes. This option also masks straggling storage servers. For long-term availability, new storage servers can be added to replace faulty ones; they can copy records from the shard's surviving servers.

All options guarantee linearizable semantics under crash failures, but they provide different trade-offs with respect to resource usage, data locality, and service recovery time after a failure. Table 3.2 summarizes these trade-offs and

compares these options with failure recovery in Corfu and vCorfu.

Handling Ordering Layer Failures

Failures in the ordering layer can affect replicas running Scalog's ordering logic as well as aggregators. Replica failures are handled by Paxos; aggregator failures are handled by leveraging the statelessness of aggregators. A storage server or an aggregator that suspects its neighboring aggregator of having failed reports to the ordering layer, which responds by creating a new aggregator to replace the suspected one. A mistaken suspicion does not harm correctness, as both the new and the wrongly suspected aggregator correctly report local ordering information to their parent.

A distinguishing feature of Scalog is that Scalog suffers no net throughput loss because of ordering layer failures. Because of Scalog's approach to decoupling ordering from data replication, storage servers continue accepting client append requests and ordering records locally in their log segments, independent of the status of the ordering layer. Any temporary loss of throughput caused by an ordering layer failure is thus made up for as soon as the failure is recovered, when these locally ordered records are seamlessly inserted in the next cut issued by the repaired ordering layer. It does cause a spike in throughput because the repair interleaves all delayed records that are already replicated in one single cut. This is in contrast to sequencer-based logs where, after throughput halts because of a sequencer failure and reconfiguration [22, 26]), throughput goes back to normal instead of compensating for the loss of availability.

3.4 Applications

Applications can configure Scalog and customize sharding policies to satisfy their requirements. This section discusses typical applications that benefit from Scalog and demonstrates how to configure Scalog and set sharding policies.

3.4.1 The Online Marketplace

The online marketplace we used to motivate the Scalog design logs user activities (sellers listing products, buyers browsing and purchasing products, etc.) to Scalog for analytics and fault tolerance. To satisfy the requirements discussed in Section 3.1, we configured Scalog to use *Finalize & Add* to handle storage server failures and for a sharding policy we let each application process write to the nearest storage server.

If an application process writes at a rate that may overwhelm a single shard, it may select multiple shards to distribute the writes. Periodically, analytics jobs read Scalog, which may negatively affect the write rate; therefore, before performing analytics jobs, the online marketplace finalizes shards and adds new shards: the online marketplace writes to newly added shards, and analytics jobs read from finalized shards. This isolation makes sure analytics reads do not negatively affect online writes.

Using the API discussed in Section 3.1, the online marketplace calls `append` to log user activities. Periodically, analytics jobs use the `subscribe` API to extract data. When any of the system components fail, the online marketplace calls `subscribe` to replay the log and reproduce its state.

3.4.2 Scalog-Store

Modeled after Corfu-Store [26], Scalog-Store uses Scalog as its underlying storage. Scalog-Store configures Scalog to handle storage server failures using *Finalize & Add*, as it is the same as how Corfu handles failures. Like the online marketplace's sharding policy, the sharding policy is for an application process to select the nearest storage server.

Scalog-Store supports the same operations as Corfu-Store: atomic `multi-get`, `multi-put`, and `test-and-multi-put` (conditional `multi-put`). Scalog-Store uses a *mapping server* with an in-memory hash map that maps each key to a pair (l, s) containing a global sequence number l and a shard identifier s where the latest record containing the value of that key is stored.

To implement `multi-get`, which takes a set of keys as input, a client retrieves, in a single atomic request, the (l, s) pairs for the keys from the mapping server. The client then calls `readRecord`(l, s) for each pair to get each key's value.

To implement `multi-put`, a client first executes `appendToShard`$(\langle key, value \rangle)$ for each key to receive corresponding (l, s) pairs. Next, the client creates a *commit record* that contains the set of $(key, (l, s))$ records for each key and uses `appendToShard` to add the commit record to the log. (An optimization for single-key `multi-put` operations is only to log a commit record containing the key and value.) The client then forwards the commit record to the mapping server, which updates its hash map accordingly and responds. `multi-put` finishes on receipt of the response. Should the mapping server crash, a new server can re-read the log and rebuild a current hash map.

The implementation of `test-and-multi-put` is similar to that of `multi-put`,

but adds a test condition to the commit record. Upon receiving the forwarded commit record, the mapping server evaluates the test condition to decide whether to commit the operation. If so, the mapping server processes the operation normally; otherwise, the mapping server processes the operation as a *no-op*. Finally, the mapping server returns the result to the client.

3.4.3 vScalog

Modeled after vCorfu [83], an object store based on Corfu, vScalog is an object store that runs on Scalog. The key difference between vScalog and vCorfu is how they guarantee data locality. vCorfu maintains a separate log, a so-called *materialized stream*, for each object, in addition to a global shared log. A client has to write an object update to the shared log for total order and to the materialized stream for data locality. vScalog, instead, leverages Scalog's sharding policy to map each object to one shard, effectively using each Scalog shard as a materialized stream. As a result, the single shared log guarantees both total order and data locality. vScalog can configure Scalog to handle storage server failures using either *Remove & Replace* or *Mask*; our implementation uses *Remove & Replace* because it is how vCorfu handles failures.

Compared with vCorfu, vScalog offers two main advantages. First, it is more robust: it can tolerate f failures in each shard, while vCorfu cannot handle a log replica and a stream replica failing simultaneously. Second, it offers higher read throughput: it lets clients read from all the replicas in a shard, while vCorfu's clients only read from stream replicas. A disadvantage is that vScalog requires all transactions, including those that will eventually abort, to be written to the

log. The fundamental reason goes back to Scalog's persistence-first architecture, as the predicate on a `test-and-multi-put` operation may depend on the position of the corresponding record in the log, which Scalog decides after the record is replicated.

3.5 Evaluation

The goal of Scalog is to provide a scalable totally ordered shared log with seamless reconfiguration. In our assessment of Scalog, we ask the following questions:

- How do reconfigurations impact Scalog? (Section 3.5.1)
- How well does Scalog handle failures? (Section 3.5.2)
- How much write throughput can Scalog achieve and what is the latency of its write operations? (Section 3.5.3)
- How well do Scalog read operations perform in different settings? (Section 3.5.4)
- How do Scalog applications perform? (Section 3.5.5)

We have implemented a prototype of Scalog in golang [8], using Google protocol buffers [10] for communication. To tolerate f failures, the ordering layer runs Paxos with $2f + 1$ replicas and each shard comprises $f + 1$ storage servers; unless otherwise specified, we set $f = 1$.

Some of our experiments use Corfu as a baseline for Scalog. To enable an "apples-to-apples" comparison, we implemented a prototype of Corfu in golang: it uses one server as a sequencer, $f + 1$ servers for each storage shard,

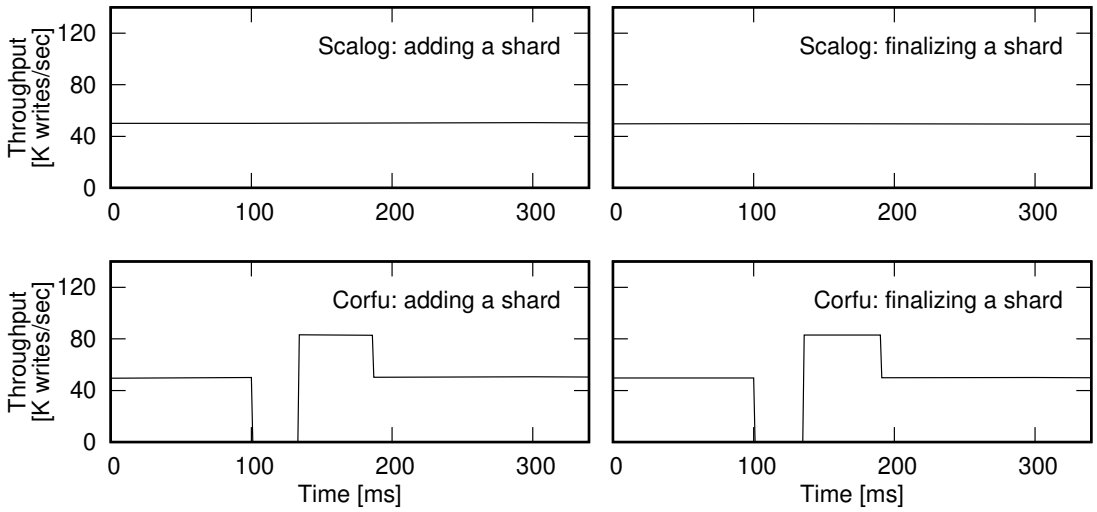


Figure 3.3: Throughput during reconfiguration

and Google protocol buffers for communication. Our Corfu implementation achieves higher throughput and lower latency than Corfu's open-source implementation [5]. To simplify comparison with published Corfu benchmarks, we fix the record size at 4KB.

We run our experiments on 40 c220g1 servers in Cloudlab's Wisconsin datacenter. Each server has two Intel E5-2630 v3 8-core CPUs at 2.40GHz, 128GB ECC memory, a 480GB SSD, and a 10Gbps intra-datacenter network connection. Since exploring the limits of Scalog's write throughput requires many more than the 40 servers available to us, we resorted to simulation for results that report on larger configurations (specifically, those in Figure 3.5 in Section 3.5.3).

3.5.1 Reconfiguration

To evaluate how Scalog and Corfu perform when shards are added and finalized, we run both with six shards, each shard having two storage servers ($f = 1$). We target 50K writes/sec, roughly half of the maximum throughput in this set-

ting. We either add a shard or finalize a shard at t = 100ms.

Figure 3.3 shows that Scalog's throughput is unaffected by adding or finalizing shards. When shards are added, clients can continue to use the original shards. Clients connected to shards are notified prior to finalization (we set the value of the configuration variable described in Section 3.3.4 to 10). During reconfiguration, throughput in Corfu ceases for roughly 30 ms because all storage servers must be notified before the new configuration can be used [26].

3.5.2 Failure Recovery

To evaluate how Scalog and Corfu perform under failure, we again deploy them with six shards, each with two storage servers, and use 50K writes/sec. To evaluate performance under aggregator failure, we add two aggregators to Scalog, each handling half of the shards. We measure throughput under four failure scenarios in Scalog: Paxos leader failure, Paxos follower failure, aggregator failure, and storage server failure, and under two failure scenarios in Corfu: sequencer failure and storage server failure. In each scenario, we intentionally kill one server at time t = 2s and measure how throughput is affected. Figure 3.4 reports the results for the six failure scenarios:

Scalog's Paxos leader and Corfu's sequencer. Although records are temporarily unable to commit, Scalog's storage servers can continue receiving new records, which are committed as soon as a new Paxos leader is elected. Hence, after a dip, throughput temporarily spikes to catch up, and total throughput is unaffected, although latency suffers until a new leader is elected. On the other hand, Corfu's clients compete for log positions when the sequencer is unavail-

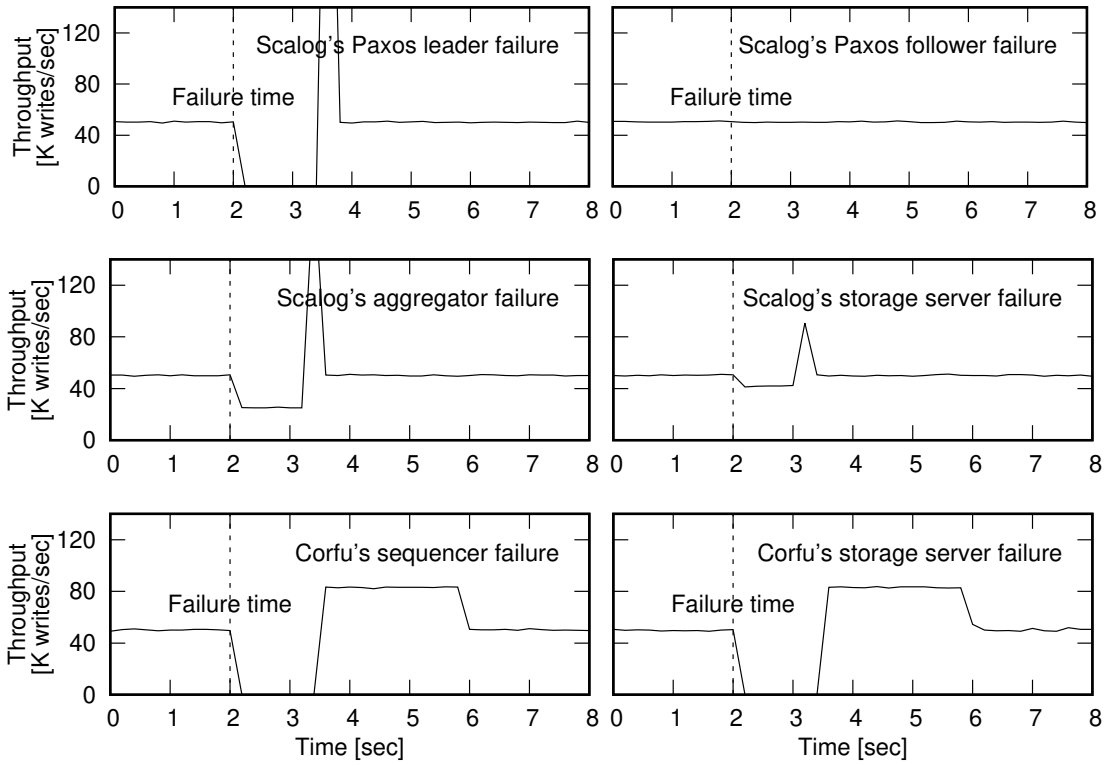


Figure 3.4: Throughput under different failure scenarios

able [26]. Heavy contention among clients causes Corfu's throughput to drop to nearly zero until a replacement sequencer joins [22]. Thereupon, Corfu runs at peak throughput until it catches up and stores all the delayed records; during this time, Corfu experiences higher latency.

Scalog's Paxos follower. No effect on throughput or latency.

Scalog's aggregator. Again, although the affected storage servers (in this case, half of all storage servers) are temporarily unable to commit new records, they can continue to receive them. Thus, the effects on throughput and latency are similar to those of a Paxos leader failure.

Scalog's and Corfu's storage servers. We compare Scalog's *Finalize & Add* with Corfu because they have the same trade-offs. In Scalog's *Finalize & Add*, the

faulty server's shard is finalized. Throughput decreases temporarily until the failure is detected (relying, in our setting, on a one-second timeout) and all clients connected to the finalized shard are redirected to storage servers in other shards. Throughput is restored after slightly more than a second. In Corfu, the faulty storage server triggers a change in the mapping function; while this takes place, Corfu is unavailable [26]. Again, once the failure recovery completes, Corfu is saturated until all buffered records are stored.

3.5.3 Write Performance

We measure Scalog's write latency and throughput by running each client in a closed loop in which it sends a record and then awaits an acknowledgment. Latency measures the time difference between when a client sends the record and when it receives the acknowledgment. Throughput measures the number of write operations per second over all clients.

Corfu's peak throughput depends on the number of shards and the sequencer's throughput. Scalog's peak throughput depends on the number of shards and the configuration of the aggregators; in addition, it also depends on the length of the interleaving interval. By increasing the interleaving interval, Scalog can increase its throughput because a higher interleaving interval allows Scalog's ordering layer to manage larger numbers of shards and storage servers at the expense of higher latency. To compare fairly against Corfu, we run our evaluation with a fixed interleaving interval, set at 0.1ms to match Corfu's write latency. As we will see in Section 3.5.3, even with this short interval, Scalog already supports many more storage servers than we have resources to deploy.

System Configuration

In both systems, as the number of shards increases, ordering becomes a bottleneck. To properly configure each system to measure its peak throughput, we run microbenchmarks to determine, (1) the maximum throughput of a single shard (using $f + 1$ storage servers) and (2) the maximum number of shards that their respective ordering layer can handle.

Throughput from one shard. A shard in Scalog peaks at 18.7K writes/sec; our implementation of Corfu, while outperforming previously reported figures for Corfu [26], reaches 13.9K writes/sec. The difference is due to how the two systems enforce total order at each storage server. In Scalog, where storage servers sequence records in the order in which they receive them, it is natural to write these records to disk sequentially. In contrast, records in Corfu are ordered by the sequencer, not by the storage servers. Records from different clients may reach storage servers out of order. Corfu storage servers first skip over missing records and later perform random writes to fix the log once those records are received.

The number of shards each system can support depends on the maximum load Scalog's aggregators can sustain and the maximum throughput of Corfu's sequencer.

Scalog's aggregators. We measure the number of shards and child aggregators that an aggregator can handle by having its neighboring servers (be they storage servers, the Paxos leader, or other aggregators) send synthetic messages. We find that each aggregator can handle either 24 storage servers (i.e., 12 shards in

our $f = 1$ setting) or 23 child aggregators, while the ordering layer can handle up to either 12 shards or 22 aggregators. We use these numbers to estimate the maximum number of shards that Scalog can support for a given number of aggregators.

Corfu's sequencer. We find that the sequencer of our Corfu implementation handles about 530K writes/sec, comparable to the optimized Corfu implementation used in Tango [27].

We want the throughput of both systems to scale linearly in the number of shards until ordering becomes the bottleneck. To avoid overloading the storage servers, we then configure each shard in Scalog and Corfu at 80% of their peak throughput, respectively, at 15.0K writes/sec and 11.1K writes/sec. To avoid overloading Scalog's Paxos leader and aggregators, we never assign to the ordering layer or to individual aggregators more than half of the maximum load they can sustain (i.e., either six shards or 11 aggregators); if the load exceeds what the system's current configuration can handle under this policy, we add a new layer of aggregators. Thus, we configure these systems as follows:

Scalog. We add one shard for every 15.0K writes/sec of throughput. With up to six shards, we do not use aggregators. Between 7 and 66 shards, we use one layer of aggregation, with one aggregator for every six shards. With more than 66 shards, we use multiple layers of aggregators, where the ordering layer handles at most 11 aggregators, each aggregator handles at most 11 child aggregators, and each leaf aggregator handles at most six shards.

Corfu. We add one shard for every 11.1K writes/sec of throughput, until the

sequencer becomes a bottleneck.

Write Scalability

We now proceed to determine how much load Scalog and Corfu can handle and, in particular, the throughput and latency that they achieve. Unfortunately, we only have access to 40 servers in CloudLab; in cases that require more servers, we emulate storage servers and their load. When communicating with the ordering layer, each (emulated) storage server reports to be receiving records at the same throughput and latency as a real storage server, though it is not receiving records from clients. This setup allows one physical machine to emulate hundreds of storage servers.

Let l_1 be the time elapsed at the client between submitting a record and learning that it is committed, and let l_2 be the time elapsed between submitting a report to the ordering layer and learning the corresponding cut. Both are measured using real storage servers. For our emulation, we use as latency the sum of (1) the time elapsed at the *emulated* storage server between submitting a report to the ordering layer and learning the corresponding cut and (2) $l_1 - l_2$.

Figure 3.5, which presents throughput/latency measurements as we increase the number of shards, shows that Scalog significantly outperforms Corfu's throughput while experiencing lower latency.

Corfu's maximum throughput is limited by the sequencer at 530K writes/sec. Emulating only storage servers, but not aggregators, with our 40 machines, Scalog reaches 2.34M writes/sec, but is still far from being saturated. To explore the limits of the workload that can be handled by Scalog's ordering layer, we

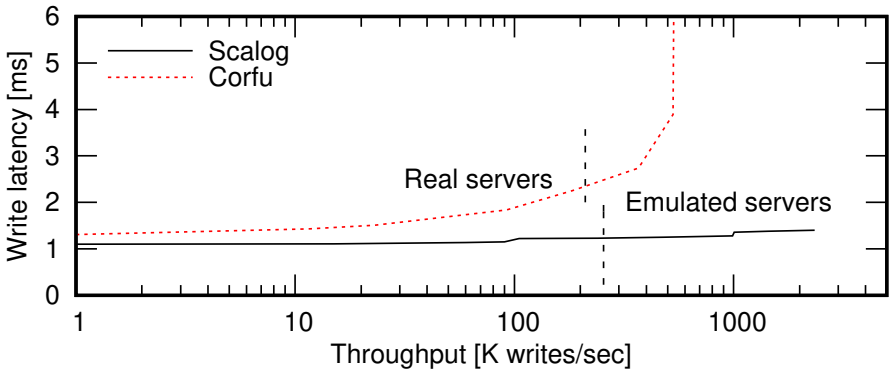


Figure 3.5: Latency vs throughput for Scalog and Corfu. The vertical dotted line separates results obtained with real servers from those obtained through emulation. For Scalog, we emulate storage servers, but not aggregators. Scalog's maximum throughput in this configuration is limited by the number of machines available to us.

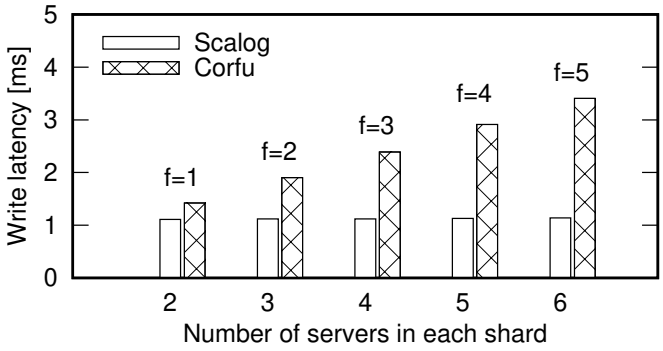


Figure 3.6: Write latency vs. shard size

deployed Paxos with multiple layers of aggregators: we used physical servers for the Paxos replicas and the uppermost layer of aggregators, and emulated additional layers of aggregators as necessary against an emulated workload corresponding to a varying number of storage servers. We found that, before Paxos becomes a bottleneck, Scalog can handle up to 3,500 shards with three layers of aggregators, which translates to 52M writes/sec.[3] This throughput could be further increased by using a larger interleaving interval, trading latency for throughput.

[3]We use emulation to measure the maximum number of shards the ordering layer can handle. We are unable to assess other scaling issues (e.g., the network bottleneck), because we do not have access to a sufficiently large testing infrastructure.

Scalog's latency in Figure 3.5 grows slightly (by about 0.1 ms) whenever a new layer of aggregators is added, but remains lower than Corfu's. Based on our experiments with one and two layers of aggregators, we estimate the latency at 52M writes/sec to be around 1.6 ms (the client perceived latency is 1.3 ms when there are no aggregators, plus three layers of aggregators at about 0.1 ms per layer).

Corfu's latency is negatively impacted by two factors: first, Corfu replicates records across storage servers using client-driven chain replication [81] that writes to each server in sequence; second, since Corfu's clients may (and, in sufficiently long runs, likely will) write records to any storage server, the overhead paid by servers in managing client connections grows with the number of clients.

Finally, we investigate how write throughput and latency are affected by f, the number of failures that a shard tolerates. We find that throughput in both Scalog and Corfu is not significantly affected when varying f; thus, we focus our discussion on latency. Figure 3.6 shows that, for a single shard, client-perceived latency in Scalog is roughly constant, while in Corfu, latency increases linearly with f. The reason is, again, that Corfu replicates a record within a shard by writing sequentially to each of its storage servers, while Scalog allows a record to be replicated in parallel on multiple storage servers. Thus, as the number of storage servers in the shard increases to tolerate higher values of f, so does the latency gap between Scalog and Corfu.

3.5.4 Read Performance

Unlike writes, reads in Corfu and Scalog follow similar paths with identical performance. We therefore only focus on Scalog's read latency and throughput.

Using a single storage server s, we measure latency with a single client and measure throughput as a function of the number of clients. To evaluate the performance of sequential reads, we have a client call `subscribe`(l), where $l \leq l_{max}$, the maximum global sequence number the storage server has observed (see Section 3.3.2). We measure latency as the time between the `subscribe` call and the receipt of the first record; for throughput, we divide the number of records between $[l, l_{max}]$ in s by the time needed to receive them. To evaluate random reads, we have a client call `readRecord`(l, s) in a closed loop, where l is randomly generated such that $l \leq l_{max}$ and record l is stored in shard s.

Normally, the client library connects to all shards for `subscribe`(l) and chooses a random server in shard s for `readRecord`(l, s) (Section 3.3.2); instead, for these measurements we modified the client library so that it connects only to the storage server in s that is the focus of our evaluation.

When the client reads data that is still stored in the memory of the storage server, the throughput for both `subscribe` and `readRecord` is 280K records/sec (i.e., the limit of a storage server's network bandwidth) and the latency for both a `readRecord` request and for receiving the first record after a `subscribe` call is about 0.09 ms.

When the client reads data that is no longer in memory (as is often the case with finalized shards), latency and throughput are limited by the performance of storage server disks. With our hardware, `readRecord` achieves 4.5K record-

s/sec throughput and 0.31 ms latency; as for `subscribe`, by reading sequentially and returning 256KB log chunks, it achieves 57K records/sec throughput with 1.21 ms latency to receive the first record; larger chunks improve throughput somewhat, but at the cost of significantly higher latency.

When a client reads from many storage servers concurrently (whether from one or multiple shards), throughput is limited by the client's network bandwidth, which is on average 280K records/sec in our evaluation.

3.5.5 Impact on Applications

We focus on the applications discussed in Section 3.4. Of these, the online marketplace uses Scalog to store user activities using `append` and reads the log using `subscribe`, so its performance is simply that of Scalog. The other two applications are more involved and deserve a more careful investigation.

Scalog-Store

We have implemented prototypes in golang using protocol buffers of both Scalog-Store and Corfu-Store based on our Scalog and Corfu implementations.

In this experiment, both Scalog-Store and Corfu-Store run on 20 storage servers (10 shards). The keys are 64-bit integers, while the values are 4088 bytes (creating 4KB records).

Figure 3.7 shows that Scalog-Store has higher `multi-put` throughput than Corfu-Store, because each storage server in Scalog has higher throughput (see

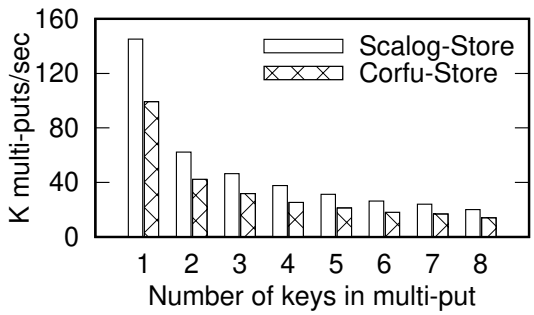


Figure 3.7: throughput of `multi-put` with 10 shards

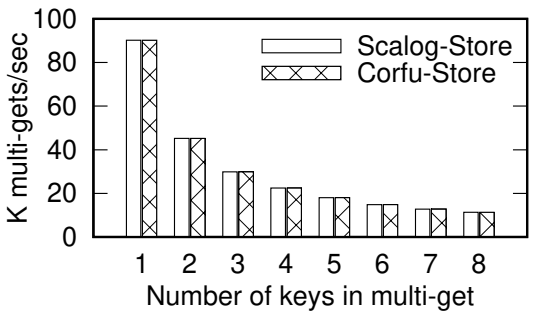


Figure 3.8: throughput of `multi-get` with 10 shards

Section 3.5.3). For both Scalog-Store and Corfu-Store, the throughput of `multi-put` operations is limited by the throughput of the log given the limited number of shards we have available. An exception is when Scalog-Store has 10 shards and one key in `multi-put`, when the bottleneck is the mapping server.

If we had many more shards but few keys in `multi-put` operations, then the mapping server would be the bottleneck for both Scalog-Store and Corfu-Store, and we would expect the `multi-put` throughput to be the same. However, if we increase the number of keys, we would expect Scalog-Store to eventually have higher throughput than Corfu-Store because the bottleneck will eventually shift to the log. This is because the throughput of the mapping server does not deteriorate much with the number of keys and the throughput that the log has to provide equals the number of keys times the throughput of the mapping server. For Corfu-Store, the shift happens when there are eight keys. Because Scalog provides superior throughput to Corfu, Scalog-Store can provide higher `multi-put` throughput when the number of keys is larger than eight.

To summarize, Scalog-Store achieves higher per-shard write throughput than Corfu-Store, because Scalog-Store uses fewer shards to achieve the same total throughput. When there are eight or more keys in each `multi-put` operation,

Corfu reaches its maximum throughput and becomes a bottleneck while Scalog does not.

For both Scalog-Store and Corfu-Store, the throughput of `multi-get` operations (Figure 3.8) is limited by the random read throughput of storage servers.

vScalog

Starting respectively from our Scalog and Corfu implementations, we prototyped vScalog and vCorfu in golang, using protocol buffers. We implemented each object as a key-value pair and ran each system as a key-value store.

We first measure the maximum write throughput of a single materialized stream, since it limits the maximum update rate of a single object. Our evaluation shows that one materialized stream of vScalog and vCorfu achieves 18.6K writes/sec and 13.6K writes/sec, respectively, which are roughly the same as the respective single shard throughputs of Scalog and Corfu shown in Section 3.5.3. The client perceived latencies for vScalog and vCorfu are 1.2ms and 1.5ms, respectively; vCorfu is slower because it writes to disks sequentially while vScalog writes to disks in parallel.

Next, we measure the total throughput of vScalog and vCorfu. Our experiments show that, using the same number of shards, vScalog has roughly the same throughput as Scalog. Using the same number of stream replicas in vCorfu as the number of shards in Corfu, and given enough log replicas, vCorfu and Corfu also achieve approximately the same throughput. However, the single shard throughput of vCorfu's underlying shared log reduces to 9.3K writes/sec (due to the cost of writing a commit bit, matching the 40% penalty reported

in [83]).

3.6 Limitations

Scalog's current prototype suffers from several limitations. Some seem to be relatively easy to address: for example, while Scalog allows applications to dynamically add and finalize shards, it does not provide automated policies to trigger such actions. Other limitations are common to storage systems that operate at a large scale: as server failures become frequent, the steps needed for recovery may complicate the scheduling and allocation of resources. Others yet, however, appear to be more fundamental to Scalog's design. In particular, although Scalog offers unprecedented throughput at latency comparable to, or better than, prior shared log implementations, it is not well suited for applications that require ultra-low latency (such as high-speed trading), highly-predictable latency and throughput, or low tail latencies. The question of whether it is possible to drastically reduce latency while maintaining Scalog's throughput and ordering properties remains open. Finally, some issues are outside of Scalog's current scope: in particular, Scalog's design does not address security concerns.

3.7 Conclusion

Inspired by crash-resistant storage systems, Scalog departs from previous implementations of the totally ordered shared log abstraction by making records persistent before determining their positions in the log. This simple but es-

sential change of perspective lets Scalog scale out elastically and recover from failures quickly; allows applications to customize which storage servers should hold their records; and enables a new ordering protocol that, by interleaving the local orders built by each storage server as a side product of replicating records, achieves almost two orders of magnitude higher throughput than the state-of-art shared log implementation.

CHAPTER 4

ZIPLOG: A TOTALLY ORDERED LOG COMBINING LOW LATENCY WITH SCALABLE THROUGHPUT

This chapter presents the design, implementation, and evaluation of Ziplog, a totally ordered shared log that combines low latency with scalable throughput.

Much recent work [14,26,83] and Chapter 3 of this dissertation have focused on matching this elegant and simple abstraction with a high-performance, fault-tolerant implementation. These efforts have shown that it is possible to reconcile the demands of total order with many of the properties of an ideal shared log, such as high scalability via multiple shards, application-friendly data layout, and reconfigurations that cause no loss in availability.

Minimizing latency while having high scalability and total order, however, has remained so far an elusive goal. For example, Scalog [38], a state-of-the-art shared log implementation presented in Chapter 3 of this dissertation, reports a 1.2 ms latency for appending a record at the end of its log; in comparison, consensus decisions in NOPaxos [67], which provides total order but not high scalability, take as little as 111 μs with the help of special hardware (comparable to the latency of a non-replicated system), and about 200 μs without.

The roots of this gap are deeper than simple engineering; as we show later in the chapter, the fundamental costs of supporting linearizable `Append` operations in a multi-shard log make low latency all but impossible.

In essence, then, developers interested in a log implementation that supports cross-shard total order, scalable throughput, and low latency must today curb their enthusiasm and pick at most two: cross-shard total order and scalable

throughput (as in Scalog [38]), total order and low latency for a single shard (as in NOPaxos [67]), or low latency and scalable throughput, but with no cross-shard total order (as in Kafka [57]).

Ziplog, the shared log that we present in this chapter, achieves all three: it guarantees total order, achieves scalable throughput, and experiences low latency. It does so by adding a fourth dimension to today's three-way tradeoff: it questions the shared log's API.

The common shared log API includes operations to query the log and append new records to the log's end. These operations generally guarantee *linearizability* [50]. Without linearizability, application developers would have to deal with the complexities of weak consistency and lack of composability. But implementing such a linearizable API requires determining where the log ends. It is the coordination needed to make this determination causes high latency, since it involves either all-shard coordination, or the mediation of some kind of ordering service (e.g., Corfu's sequencer [26] or Scalog's ordering layer [38]).

Ziplog adopts a new API that "cuts out the middleman" without requiring all-shard coordination: it adds a new record R to the log by involving only a single shard; perhaps surprisingly, it can do so without giving up linearizability. Concretely, instead of adding a new record R to the end of the log, Ziplog's `InsertAfter`(R, rid') allows clients to specify the entry in the log (identified by the record identifier rid') past which R should be inserted; the operation, once it completes, returns the record identifier assigned to R. For queries, clients can either request a specific record by specifying its record identifier or subscribe to the log starting from a record identifier to retrieve a sequence of (R, rid) pairs.

These operations are linearizable, and, although they have different semantics than the more common shared log API, they still allow building applications with meaningful consistency guarantees easily. For example, clients know the identifier of every record they previously inserted, and, after subscribing to the log, learn the identifier *rid'* of every record *R'* they retrieved. Thus, by invoking `InsertAfter`(*R*, *rid'*), it is easy for them to ensure that any record *R* causally dependent [60] on *R'* appears in the log after *R'*.

Ziplog's `InsertAfter` implementation relies on a new consensus algorithm that, in the absence of contention, requires only two message delays to totally order a new record and add it persistently to the log, at a latency of about 150 μs; contention only adds one message delay, bringing Ziplog's latency to about 220 μs. In practice, we find that Ziplog's average latency is lower than what NOPaxos can offer without help from special hardware.

Ziplog requires cross-shard coordination only to deal with failures and reconfigurations: these events are handled through Paxos-based services. Ziplog's reconfigurations are seamless [38]: the coordination takes place outside of the clients' critical path, and they experience no additional latency. The failure of a storage server temporarily suspends `InsertAfter` operations only in that server's shard; however, existing subscribe operations delay returning records until the affected shard is once again fully operational.

Our evaluation confirms that Ziplog manages to offer an unprecedented combination of features: total order, high throughput, low latency, seamless reconfiguration, and a linearizable API.

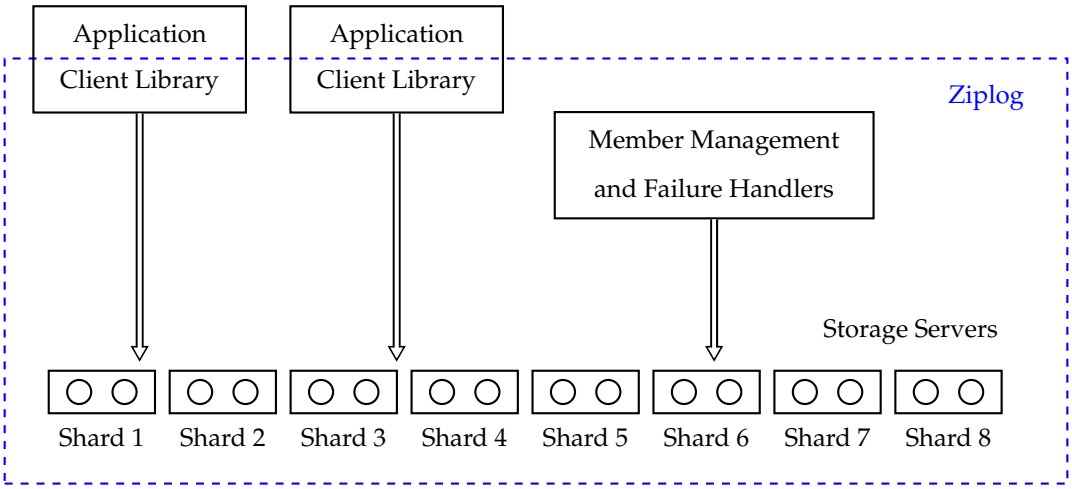


Figure 4.1: Ziplog's architecture

4.1 Ziplog's Goals and Non-goals

Ziplog's goal is to achieve throughput comparable to Scalog's and latency comparable to that of NOPaxos, without compromising either total order or seamless reconfiguration, and without using special hardware. It is not the aim of this Ziplog prototype to achieve the lowest possible latency given the current technology. Several engineering techniques that the current prototype does not use, such as RDMA [18] and DPDK [6], could reduce the latency caused by the TCP stack and increase throughput. These techniques are orthogonal to the design of Ziplog: they can apply to all shared log implementations and increase their throughput and decrease their latency. Quantifying their impact is outside the scope of this chapter. Instead, Ziplog aims to achieve its goals through a novel design motivated by diagnosing the root causes of high latency in today's totally ordered, high throughput shared logs.

`SetPolicy`(*p*)	Set the sharding policy that specifies which shard a record is written to.
`InsertAfter`(*R*, *rid*)	Insert record *R* to the log, guaranteeing that it is after the record with identifier *rid*. Returns the record identifier for *R*.
`ReadRecord`(*rid*)	Read the record with identifier *rid*.
`Subscribe`(*rid*)	Subscribe to the log, starting with the record with identifier *rid*.
`Trim`(*rid*)	Delete all records before the record with identifier *rid*.

Table 4.1: Ziplog API

4.2 Design and Implementation

Ziplog is designed to work in a datacenter, with a large number of application servers (the clients of Ziplog) and a cluster of storage servers (see Figure 4.1). Application servers may co-locate with storage servers, depending on resource management policies and application requirements.

Ziplog assumes a crash failure model with asynchronous communication channels: servers may crash or stop to respond for an indefinitely long time, and message delivery time has no upper bound. Accurate failure detection is impossible [32, 44]. Ziplog does assume that local clocks run at approximately the same rate for good performance, although it does not need it for correctness.

Ziplog partitions storage servers into a collection of shards. Each shard consists of $f + 1$ storage servers, where f is the number of failures each shard tolerates. In this setting, each shard guarantees read availability despite f failures. For write availability, Ziplog follows the same approach as Corfu and Scalog: a shard with at least one failure is not available for writing, but the shard may be *finalized* and replaced with another to guarantee the write availability of the log as a whole (see Section 4.2.5).

4.2.1 Ziplog API

Table 4.1 shows the Ziplog API. The `InsertAfter` method adds a new record to the log and returns a *record identifier* for the record. It guarantees that the new record is inserted after the one whose identifier is passed as `InsertAfter`'s second argument. The API defines a constant record identifier `RID_GENESIS` corresponding to the start of the log, which may be used if the location for the new record is not important. The `SetPolicy` method lets clients select in which shard the record will be stored (the same facility exists in vCorfu [83] and Scalog). The `ReadRecord` method returns the record for a given record identifier. The `Subscribe` method allows applications to sequentially read the log starting from the record whose *rid* is passed as an argument, and wait for future records. `Subscribe` can also accept two special record indicators, `RID_GENESIS` and `RID_RECENT`: the first causes `Subscribe` to read the log from the beginning; the second, to start reporting from a recently inserted (but not necessarily the latest) record. Applications can perform garbage collection using `Trim`, which deletes all records stored before a specific record identifier.

4.2.2 Design Overview

Because one of Ziplog's goals is to minimize latency, we keep the critical path of each operation (the code that all operations must execute) as simple as possible. To insert data, the client library first applies the current sharding policy set by `SetPolicy`(*p*) to choose a shard. Ziplog chooses a random shard if no policy has been specified. Then, it sends a record to the shard; the shard replicates the record and assigns it a *global sequence number* or *gsn*, which is returned to the

client library. Finally, the client library returns to the client a record identifier, which, opaque to the application, is a tuple consisting of its global sequence number and the shard identifier that stores the record.

Ziplog's protocol for adding records consistently and durably to its totally ordered log thus relies on three main functionalities:

GSNA (Section 4.2.3) The Global Sequence Number Assignment algorithm lets each shard independently (i.e., without cross-shard coordination) assign a *gsn* to the record. The global sequence number assignment algorithm must adapt whenever the set of shards in Ziplog changes: it must be able to allocate *gsn*s to newly added shards and, similarly, it must stop allocating *gsn*s to removed shards.

Replication (Section 4.2.4) Ziplog's new consensus protocol allows a shard's storage servers to agree very quickly on the *gsn* order of the records they replicate, thus contributing to Ziplog's goal of reducing latency.

Failure recovery (Section 4.2.5) Ziplog uses the logically centralized services of a *failure handler* to recover consistently from the failure of storage servers within a shard.

To read a record, given a specific record identifier, the client library sends the *gsn* in the record identifier to the shard specified in the record identifier. The shard finds the record in its local storage and sends it back to the client.

To subscribe to the log, the client library sends a $\langle \texttt{SUBSCRIBE}, g \rangle$ message to every shard (where g is the *gsn* stored in the record identifier passed to the

`Subscribe` API). Each shard responds by sending back all records that it stores with *gsn*s higher than g. The client library collects these records and presents them to the client in increasing *gsn* order, without skipping over any *gsn*. If the client library encounters a missing *gsn*, it must hold off processing records with larger *gsn*s until it receives the missing record. Unfortunately, this delay increases the *sequential access latency*: the time elapsed from when a client invokes `InsertAfter`(R, *rid*) to when R is returned to a client who has invoked `Subscribe`. Minimizing this latency is one of Ziplog's key challenges.

4.2.3 Global Sequence Number Assignment

The key to Ziplog's low latency is a design that empowers shards to individually assign global sequence numbers to records, while still guaranteeing that (*i*) no global sequence number belongs to more than one shard, and (*ii*) each global sequence number belongs to some shard. We explained the basic idea in Chapter 1. There are two challenges when turning this idea into practice. The first challenge is that the set of shards may change because of failures or reconfigurations. The second challenge is that different shards may see different rates of updates: a Global Sequence Number Assignment (GSNA) algorithm that does not account for this disparity may lead clients to experience latencies when accessing the log sequentially.

To illustrate the second challenge, consider a system with two shards, S_0 and S_1, and assume they receive records at approximately the same rate; a simple GSNA might have shard S_0 assign even *gsn*s and S_1 assign odd *gsn*s. This would work well if the two shards receive records at approximately the same rate.

However, if rates are different, say, records on S_0 are written at five times the rate of those on S_1, then clients who subscribed to the log would have to wait for odd-numbered slots to fill, though even-numbered positions further in the log are already filled. To prevent this, less-in-demand shards could proactively fill their unused slots with no-ops, but this approach may waste the significant space and communication bandwidth used to store and retrieve these no-ops.

The Ziplog *membership management service* (MMS), which is replicated for fault tolerance using Paxos, addresses both challenges. The role of this service is twofold. First, it provides each shard with up-to-date information about the set of shards that can be used to insert new records in the log. Second, it makes it possible to think of Ziplog as executing through a sequence of *epochs*, where the length of each epoch is defined, at each shard, by the time interval between two successive broadcasts from the MMS. Each broadcast starts a new epoch by sharing with all shards four items:

1. the identifiers of the shards that are accepting new records in the current epoch;
2. the rate of updates that each shard expects to experience during the current epoch;
3. the *base gsn*, which is the first *gsn* assigned to this epoch; and
4. the *ending gsn*, which is the largest *gsn* that may be used by this epoch.

New epochs are triggered either (i) by changes in the set of shards that accept inserts, or (ii) to periodically adjust the shards' expected rate of updates. Log continuity is ensured by making the base *gsn* of the next epoch equal to the ending *gsn* of the current epoch, plus one.

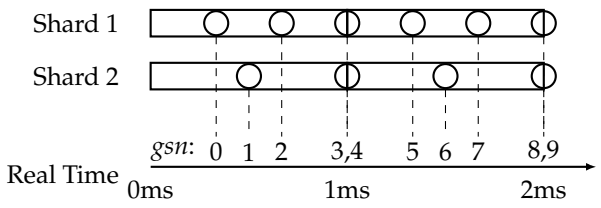


Figure 4.2: Example of *gsn* assignment. Record arrival rates at Shard 1 and Shard 2 are 3 records/ms and 2 records/ms, respectively. Circles, representing the expected arrival time of each record in each shard, are projected onto the real-time axis to produce a total order.

Armed with this information, shards can independently assign a unique *gsn* to each record they receive while avoiding the pitfalls of our simple two-shard example. In particular, each shard can compute the expected arrival time of records at every other shard, and assign *gsn*s to its records in a way that reflects that order, using shard identifiers to break ties (see Figure 4.2). In principle, a shard can use this approach to compute all the *gsn*s that it will be allowed to use in the current epoch: in practice, shards only compute *gsn*s as they need to.

When record arrival rates for the current epoch are accurate, the log can be filled without no-ops and without delays in sequential log access; but what if they are not? Each shard can compute the time by which it expects the next record. If none occurs past a certain timeout, the shard assigns the corresponding *gsn* to a no-op. However, if records are received at a higher rate than expected, shards simply assign to them their next *gsn*s for this epoch. Shards share the update rate experienced in the current epoch with the MMS, which uses it to set the rate value for the next epoch.

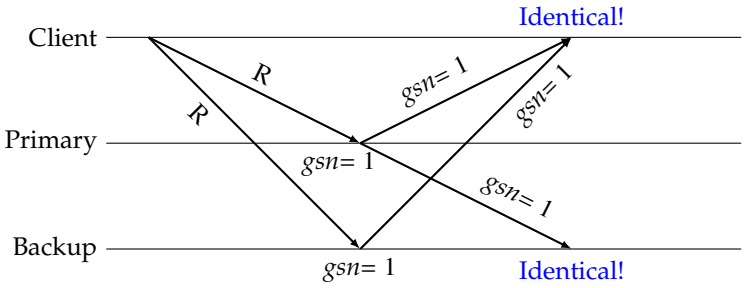
We tune the length of an epoch so that shards will not run out of *gsn*s. Epochs are identified by a sequence number e that is incremented by one with each new epoch. In our current implementation, we use 64-bit integers for *gsn*s. We set the base *gsn* to $e \times 2^{32} + 1$ and the ending *gsn* to $(e + 1) \times 2^{32}$. We currently set

the length of each epoch to 10 ms. This interval is short enough to ensure that shards will not exhaust their quota of *gsn*s and yet large enough to not introduce significant overhead. At the same time, it allows us to quickly address changes in configurations and update rates.

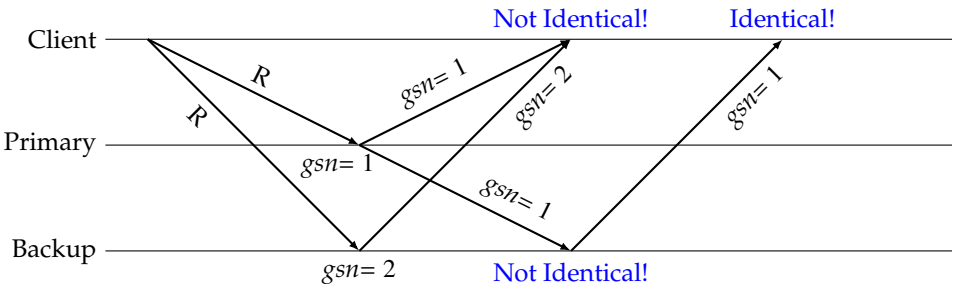
The transition between two epochs requires special care. When an epoch ends, shards are likely to be left with a very large number of unused *gsn*s. Rather than going through the expensive step of calculating each of their own *gsn*s, each shard creates a single special no-op token covering every *gsn* in the epoch, whether their own or destined to a different shard, past the last *gsn* they have used.

This approach introduces a subtlety: it is possible for the shard that owns a *gsn* to have associated it with a record, while other shards have instead associated it with the special no-op token. Therefore, when a client library, having subscribed to the log, receives from a shard a special no-op token covering a range of *gsn*s for epoch *e*, it must wait to receive the special no-op token from every shard in *e* before it can determine for which of these *gsn*s it should return a no-op.

New epochs can also be triggered by reconfigurations. To add a new shard, the MMS lists it in its next broadcast, together with its expected rate of updates. When a shard is about to be removed, it notifies its clients, directing them to submit any future record to another shard. Once they have done so, the shard reports to the MMS an expected update rate of zero for the next epoch: this tells the MMS to remove this shard from its next broadcast. The shard will henceforth no longer accept new records, but will continue to be available for read operations.



(a) The client receives identical *gsn*s



(b) The client initially does not receive identical *gsn*s and has to wait

Figure 4.3: Ziplog's Replication Protocol without and with contention

4.2.4 Replication

Ziplog's novel design reduces the problem we started with—assigning to a new record a globally unique sequence number in a totally-ordered multi-shard log—to a more familiar problem: that of running consensus among the replicas of a single shard to ensure they agree on the record's *gsn* and store it in a fault-tolerant way. The next design challenge is then how to accomplish this while minimizing latency.

A natural approach would be to simply incorporate NOPaxos [67], today's best-of-breed. NOPaxos reports latencies as low as 111 μs, but with the help of special hardware and assumptions about the network (such as support for multicast) that do not typically hold in today's cloud platforms [35]. Even without these assumptions, running consensus in NOPaxos requires only three message

delays, at a latency of 200 μs.

Ziplog's own novel consensus protocol, outlined below, shows that it is possible to do better. In the absence of contention, consensus requires only two message delays and about 150 μs; contention adds only one message delay and about 70 μs.

The protocol. Each shard in Ziplog stores a subset of the records. A shard consists of a collection of storage servers, one acting as the primary. To insert a record R into the log, the client library selects a shard based on the sharding policy and broadcasts the record to all storage servers of the shard. Each storage server (*i*) speculatively assigns a global sequence number to the record (see Section 4.2.3), (*ii*) sends to the client library an acknowledgment containing the *gsn* it has independently calculated, and (*iii*) makes the record persistent in log position *gsn*. If the client library receives identical *gsn*s from all storage servers, it can conclude that the record is ordered, and the insert operation completes after two message delays (Figure 4.3a).

What if it doesn't? Then, the client continues waiting (Figure 4.3b), while the protocol takes further steps to ensure that all replicas eventually store R in the same log position gsn_p as the shard's primary replica.

Specifically, the primary, having sent its gsn_p to the client, concurrently forwards the tuple $\langle R, gsn_p \rangle$ to the other storage servers in the shard (the backups). When a backup receives this tuple from the primary, it checks whether the record it is currently holding in log position gsn_p is indeed R. If so, no further action is needed; if not, the backup stores R in log position gsn_p and sends an updated acknowledgment to the client that initiated R.

If in this process the backup overwrites some record R' it already held at log position gsn_p, no harm is done: the client that issued R' could not have concluded that Ziplog had given R' log position gsn_p, since it could not have received an acknowledgment for that log position from the primary.

This simple protocol ensures that, in the absence of failures, primary and backups will eventually store identical records in corresponding log positions, and clients that want to insert a record will receive matching acknowledgments after at most three message delays (Figure 4.3b).

Note that a storage server sends an acknowledgment to the client *before* writing the record to its disk. We believe this is a reasonable design choice in modern datacenters because (*i*) in case of uncorrelated failures, at least one storage server survives that has the record, and (*ii*) in case of correlated failures such as power outage in a datacenter, auxiliary batteries such as a distributed Uninterruptible Power Supply (UPS) effectively make DRAM durable: they give storage servers enough time to flush records to disks [39]. Besides reducing latency, delaying making records persistent to disk also allows storage servers to batch records before writing them to disk, improving overall throughput.

To read a record given a record identifier, the client library sends a request containing the record's *gsn* to each storage server in the shard. Each storage server that has a corresponding record responds to the client. The client library needs to check that it received identical responses from all storage servers. If not, the client library continues waiting, as before. When a storage server receives an updated *gsn* from the primary, it notifies the client library. In the absence of failures, the read operation is guaranteed to complete.

While this protocol achieves the minimum of two message delays to read a record, it is expensive and does not work in the face of failures. Ziplog's **stable records confirmation protocol** allows the client library to retrieve the record from a single storage server, instead of requiring it to hear from all of them. To achieve this, each backup sends periodically to the primary the highest *gsn* of the longest prefix of consecutive records received from the primary and stored in its local storage. The primary collects this information and periodically sends the backups the minimum among those values, which is the number of consecutive records on which all storage servers agree. We call such records *stable*—they can no longer be lost or re-ordered because of storage server failures. When responding to a read request from the client library, the storage server includes whether the record is stable or not—only if it is *not*, does the client library need to get a matching response from all storage servers; otherwise, a single response suffices. If a storage server fails, Ziplog relies on the protocol discussed in Section 4.2.5 to ensure that all reads eventually complete.

4.2.5 Failure Recovery

When a storage server fails, the shard that the failed storage server belongs to is temporarily unable to process insert requests (though it remains available for historical read operations). Ziplog uses *failure handlers* to deal with suspected storage server failures. Like the member management service, a failure handler is a logically centralized entity, replicated for fault tolerance using Paxos. Each failure handler manages failure recovery for a subset of shards. The number of shards a failure handler can manage depends on the failure rate in the cluster.

Failure recovery in Ziplog involves three phases: (*i*) failure suspicion; (*ii*) state acquisition; and (*iii*) state recovery.

Failure suspicion. Storage servers within a shard keep track of each other's health by periodically exchanging heartbeat messages. Storage servers report suspected failures to their failure handler. Because each shard has $f + 1$ servers and we assume at most f of them may fail, at least one storage server will report such suspicions.

When a storage server s suspects that one or more of its peers in the shard have failed, it contacts the failure handler to initiate the recovery process. First, the storage server promises the failure handler that it will no longer store new records (unless directed to do so by the failure handler). By freezing s in its current state, this promise ensures that records stored by other storage severs in log positions that are unfilled at s cannot be or become stable, since any such record, to become stable, needs to be stored and acknowledged by s. Therefore, the failure handler can store any record (or a no-op) in these log positions during failure recovery.

Following this, s and the failure handler engage in the next phase of failure recovery: state acquisition.

State acquisition. The purpose of this phase is to allow the failure handler to acquire and persistently store s's current state, and to use it as the basis for initializing the recovered shard to a consistent state. In particular, s shares with the failure handler the state of each of its log positions, which can be either (*i*) filled and known to be stable (see the stable records confirmation protocol in Section 4.2.4); (*ii*) filled (may or may not be stable); or (*iii*) unfilled.

Records stored in filled and stable log positions are safe: assuming at most f failures per shard, they cannot be lost, although in practice, it may still be desirable to re-replicate them during the next phase of recovery to regenerate their $f + 1$ copies within the recovered shard.

Records held in log positions that are not marked as stable need to be handled more carefully: although s does not know these records to be stable, they may well be, and indeed clients may have already become aware of that. Thus, failure recovery cannot result in storing a no-op or some different record in these log positions; instead, recovery must necessarily *roll forward*, and ultimately ensure that the records s currently holds are replicated at all $f + 1$ storage servers in the recovered shard.

Finally, records held by any of the shard's storage servers in log positions that are unfilled at s cannot be currently stable (nor can they become stable, because of s's earlier promise to the failure handler). Thus, during state recovery, these log positions can be safely filled with no-ops.

Note that s could fail during the state acquisition phase; it would be unsafe for the failure handler to start roll-forward recovery while relying on an incomplete storage server state. Since a shard can experience at most f failures, failure recovery is guaranteed to terminate: at least one of the shard's storage servers will detect the failure of the others and manage to complete the state acquisition phase.

State recovery. After having completed the process of acquiring and persistently storing the state of a storage replica, the failure handler informs the MMS. The MMS temporarily removes the shard from the configuration and responds

to the failure handler. The response includes all epochs from the one included in the report to the last one that includes the shard.

After restarting or replacing failed storage servers, the failure handler uses the information from the MMS to apply roll-forward recovery to the records that during state acquisition were stored in filled log positions, whether stable or not (unless the corresponding log positions have been trimmed). The remaining log positions are filled with no-ops. When finished, the failure handler notifies the MMS, which then re-inserts the recovered shard in the configuration using its original shard identifier.

With the current design, a record may appear in the log more than once. To wit, assume that some storage server s receives record R from a client and stores it at log position i. Next, it receives the same record from the primary at log position j, $j \neq i$. Finally, suppose all other storage servers in the shard crash. If the failure handler cannot determine which log position stores a stable copy of the record, then it must keep both.

Ziplog currently relies on applications to filter out duplicate records. This issue, however, can be resolved within Ziplog by adopting the following rules:

- all records are uniquely identified by the client libraries that send them, for example using a pair (client identifier, sequence number);
- all storage servers reject duplicate records received from client libraries (they must do so in any case to avoid record duplication in the log due to retransmissions);
- backup servers, upon receipt of a record R from the primary for a slot j,

remove any other copies of R in other slots before storing R in slot j.

This solution does not come without overhead, and thus *exactly-once* delivery will be provided optionally in a future version of Ziplog.

4.3 LogDB

To demonstrate the utility of shared logs and to evaluate the importance of the combination of high scalability, high throughput, and low latency, we have designed and implemented *LogDB*. LogDB is a transactional key/value store that uses a shared log as the storage layer. It implements transactions using optimistic concurrency control (OCC) [58].

Besides the shared log, LogDB includes a transaction coordinator and a client library. The shared log is used to store key/value mappings and also to record and order transactions. The transaction coordinator stores a mapping from each key to the latest record identifier in the log and is responsible for deciding whether a transaction should commit or abort. The client library interacts with both the shared log and the transaction coordinator to submit transactions.

In each transaction, a client can interactively read and write keys using the client library. To read the value of a key, the client library first interacts with the transaction coordinator to retrieve the record identifier, then reads the shared log to retrieve the value. To update a key/value pair, the client library inserts a record in the log and stores the record identifier. At the end of the transaction, the client library writes to the log a *commit record*, which includes the read set and the write set for all keys involved in the transaction and their corresponding

record identifiers.

The transaction coordinator, which subscribes to the log, attempts to execute the transaction when it sees the commit message. The transaction coordinator checks the read set and the write set to detect read-write and write-write conflicts: if the record identifier of a key in the read set is different from that in the mapping it stores or the record identifier of a key in the write set is before than that in the mapping, the transaction aborts; otherwise, the transaction commits and the transaction coordinator modifies its mapping based on the write set. Finally, the transaction coordinator sends the transaction decision to the client library.

Our implementation of LogDB can use either Scalog or Ziplog as the storage layer. In the Scalog version of LogDB, clients use the `appendToShard` interface to append write operations and the commit message to random shards of the log. Record identifiers are pairs of shard identifiers and local sequence numbers. To read from the log, clients call the `readRecord` interface. In the Ziplog version of LogDB, clients use the `InsertAfter` interface to insert write operations and the commit message to random shards of the log. To ensure that the transaction coordinator sees all the operations the commit message depends on, the commit message must be inserted after the largest record identifier of all read and write operations in the transaction. Note that Ziplog's `InsertAfter` interface is sufficient.

The transaction coordinator stores its state in memory and is not replicated. If it fails, a new transaction coordinator can replay the log to reconstruct the state. To reduce this reconstruction time, the transaction coordinator could periodically store a snapshot and compact the shared log using techniques like the

segment cleaner in log-structured file systems [75].

4.4 Evaluation

Ziplog seeks to achieve total order, scalable throughput, and low latency. As discussed earlier in this chapter, other shared log implementations pick at most two of these properties. As total order is a binary property, we only need to consider two baselines: totally ordered shared logs that optimize either latency or scalable throughput. For the former, the state-of-the-art is NOPaxos [67], while for the latter it is Scalog [38].

In particular, we consider the following questions:

- Can Ziplog achieve low latency, competitive with NOPaxos? (Section 4.4.1)
- Can Ziplog achieve scalable throughput, competitive with Scalog? (Section 4.4.2)
- What is the performance of read operations in Ziplog? (Section 4.4.3)
- What is the impact of using the `InsertAfter` semantics? Does it affect Ziplog's subscribe operations under dynamic workloads? (Section 4.4.4)
- How do reconfigurations and failures impact Ziplog? (Section 4.4.5)
- How do Ziplog applications perform? (Section 4.4.6)

We implemented Ziplog, NOPaxos and Scalog using the same codebase. Ziplog is designed for cloud computing and serverless computing: because today's cloud platforms do not support multicast nor allow implementing the NOPaxos sequencer in a programmable switch [35], the sequencer in our NOPaxos implementation is a server (i.e., NOPaxos's *end-host sequencing configuration* [67])

and uses multiple TCP connections to broadcast messages. Because Ziplog writes records to disks asynchronously (Section 4.2.4), our Scalog and NOPaxos implementations do the same. We used 4KB records, consistent with published results for Scalog [38] and Corfu [26].

To tolerate f failures, each shard has $f + 1$ storage servers, while the membership management service and failure handlers each runs Paxos with $2f + 1$ replicas. In the evaluation, unless otherwise specified, we use $f = 1$ (as did Scalog).

We ran our experiments on c220g1 servers in CloudLab's Wisconsin datacenter [4]. Each server has two Intel E5-2630 v3 2.4GHz 8-core CPUs, 128GB ECC memory. The network supports 10Gbps (Scalog used the same setup in their evaluation [38]).

4.4.1 Write Latency

We evaluate write latency and compare it with Scalog and NOPaxos by configuring both Ziplog and Scalog with six shards. Given the small number of shards, we disabled Scalog's aggregation layer for improved performance. NOPaxos does not support shards.

Ziplog has two execution paths in the case when there are no reconfigurations or failures: a "fast path" of two message delays (when there is no contention) and a "slow path" of three message delays (when the primary resolves contention). We first run microbenchmarks to understand the average write latency of both paths. The write latency is the time from when a client invokes

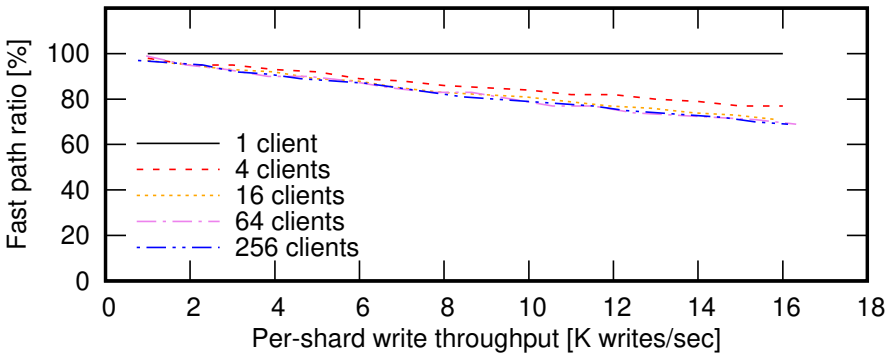


Figure 4.4: Ziplog's fast path ratio vs. per-shard throughput

the `InsertAfter` interface until it returns. The experiment shows that the fast path takes roughly 150 μs on average, and the slow path takes 220 μs (both with negligible variance). Unsurprisingly, the slow path latency is similar to the latency of NOPaxos with end-host sequencing as both involve three message delays.

Under a realistic workload, some percentage of requests will use the fast path and some the slow path. The latency distribution of Ziplog is significantly affected by this ratio. Several factors can cause contention that increases the use of the slow path. In the next experiment, we vary the per-shard write throughput as well as the number of clients per shard. For a target throughput of x writes/sec and n clients, each client adds records to the shared log at x/n times within each second chosen uniformly at random.

Figure 4.4 presents the ratio of operations that complete with the fast path. It shows that, with an increasing number of clients or increasing throughput, the ratio decreases until either the shard saturates at a throughput of roughly 16.8K writes/sec or there are more than 16 clients per shard. In the worst case, with more than 16 clients and maximum throughput, the ratio is roughly 70%. The explanation for this high ratio is as follows. Using a 4KB record size, each

shard achieves roughly 16.8K records/sec throughput, bottlenecked by SSDs. Thus `InsertAfter` operations are separated by 60 μs on average. However, latency in our experimental environment roughly follows a normal distribution with $\sigma < 10$ μs, much lower than the average separation between operations. Therefore, the chance of multiple `InsertAfter` operations happening at the same time, causing contention, is low.

Finally, we investigate the average write latency for various target write throughputs for Ziplog using one (no contention) and 256 (significant contention) clients per shard, and compare them with Scalog and NOPaxos. We run Scalog and NOPaxos with a single client in an open loop, varying the target write throughput and measuring the average latency. Figure 4.5 presents the per-shard throughput versus (average) write latency. Ziplog achieves a write latency of 150 μs at low contention (one client and low throughput) and of 200 μs at high contention (256 clients and high throughput), while NOPaxos's latency is between 220 μs and 240 μs (matching the latency of end-host sequencing in the NOPaxos paper [67]). Scalog achieves 900 μs latency, which is better than the figure published in the Scalog paper [38] on the same hardware because we modified the Scalog code to write to disks asynchronously.

4.4.2 Scalable Write Throughput

As shards in Ziplog are mostly independent of one another, the maximum throughput of Ziplog is the throughput of a shard times the maximum number of shards it can support. Figure 4.5 shows that a single shard can achieve approximately 16.8K writes/sec. To determine the maximum number of shards Ziplog can sup-

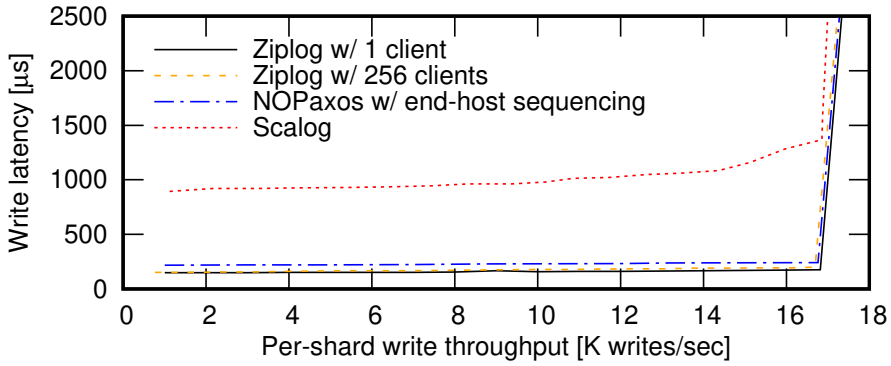


Figure 4.5: Average write latency vs. per-shard throughput

port, we ran microbenchmarks on Ziplog's membership management service. The number of shards that Ziplog can support depends on how frequent the membership management service broadcasts membership information. With a 10 ms broadcast interval, Ziplog can support about 4,000 shards, slightly higher than the number of shards Scalog can support [38]. The broadcast interval does not affect write latency; however, for reasons discussed in Section 4.2.3, it does affect sequential access latency (Section 4.4.3).

4.4.3 Random Read Performance

Ziplog's `ReadRecord` interface reads a record at a random location. Using a single shard, we measure latency with a single client. To learn the maximum throughput, we increase the number of clients. When clients read records that are stored in storage servers' memories, network throughput is the bottleneck; when the records are on disk, the bottleneck is disk throughput.

There are two cases when reading a record from memory. For recent records, a client must query all storage servers in the shard and receive responses from all of them. Though the latency, at about 100 μs, is similar to Scalog's latency,

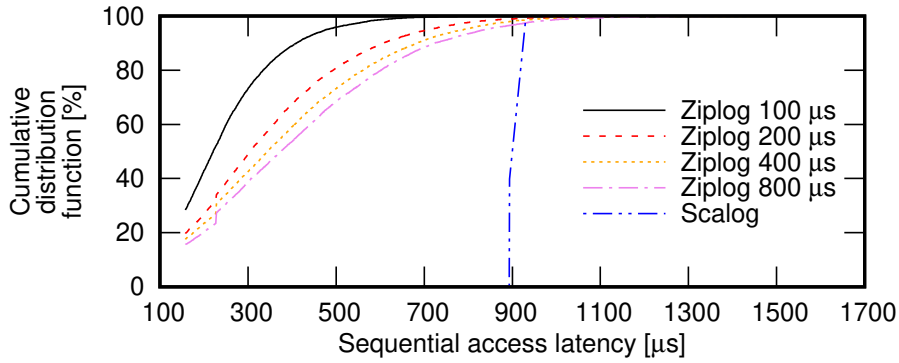


Figure 4.6: Sequential access latency with dynamic workloads for varying the average duration between operations in each client compared to Scalog.

the throughput at 280K records/sec is only half the throughput of one shard in Scalog because in Scalog clients can concurrently read from all storage servers in the shard. On the other hand, reading historical records in Ziplog can be done in parallel just like in Scalog (due to the stable records confirmation protocol presented in Section 4.2.4)—the throughput and latency are the same as Scalog's. When reading from disk, Ziplog achieves 4.5K records/sec and 330 μs latency, matching the performance of Scalog.

4.4.4 Subscribe Performance

Ziplog's `Subscribe` interface starts a sequential read of a series of records. Subscribing from historical records in Ziplog has the same workflow as that in Scalog and achieves the same throughput and latency. For recent records, clients in Ziplog may experience temporary holes in the log, while Scalog clients always receive records in order. A hole prevents records ordered after the hole from being processed and increases the sequential access latency, the time from when a client adds a record to the log to when another client can sequentially access the record.

We first ran an experiment to understand how variance in the workload affects sequential access latency. We configured Ziplog with six shards. Each client called the `InsertAfter` interface every t μs in an open loop, where t follows a uniform distribution $\mathcal{U}(0, 2T)$. A larger T means a higher variance in the workload. For a given T, we increase the number of clients until each shard processes 10K records/sec. Figure 4.6 presents the distribution of sequential access latency for varying T. As expected, higher variance in the workload results in higher sequential access latency. Overall, Ziplog experiences lower average sequential access latency than Scalog, but has a long tail when the variance is high.

Next, we ran an experiment to understand how changes in the workload affect Ziplog's sequential access latency. In the experiment, each shard initially runs at 10K records/sec, and then we instantly increase the throughput of S_1 (one of the shards) to 15K records/sec. Reading records in S_1 experiences a linearly increasing sequential access latency until storage servers receive an updated write rate from the membership management service, which takes about 1500 μs. This includes about 1000 μs for S_1 to detect the throughput change and about 500 μs for updating the write rate. The sequential access latency of reading records in other shards is not affected. We also ran an experiment decreasing the throughput of S_1 from 10K records/sec to 5K records/sec. As expected, we found that sequential access latency is not affected because S_1 fills holes in the log with no-ops.

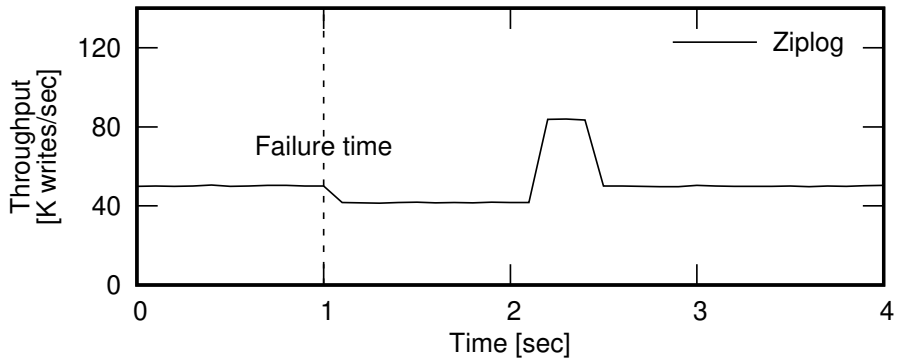


Figure 4.7: Throughput during failure recovery

4.4.5 Reconfiguration and Failure Recovery

To evaluate how Ziplog performs under reconfiguration and failure recovery, we ran Ziplog with six shards and 48 clients. To be able to compare against Scalog, each shard connected to eight of the clients, and each client added records at 1.04K writes/sec to obtain a total throughput of 50K writes/sec, the same as the setting used in the Scalog paper [38].

As discussed in Section 4.2.3, a shard notifies its clients to redirect to other shards before sending the removal request to the member management service. Through evaluation, we find that Ziplog's throughput is unaffected by adding or removing shards.

At time t = 1s, we simulated a failure by killing one of the storage servers. Figure 4.7 shows that Ziplog's write throughput temporarily decreases (by $1/6^{th}$) until the failure is detected and all clients connected to the failed shard are redirected to other shards. The failure timeout in our setting is 1s. As in Scalog, write throughput is restored slightly more than a second after the failure occurred. However, while failures in Scalog do not affect clients that read from the log through other shards, unfilled holes in the Ziplog log cause additional

delays until the failure recovery completes. This takes shorter than two seconds, including one second for failure detection and the rest for the failure handler to complete the failure recovery protocol.

4.4.6 Impact on LogDB

To understand the effect of lower latency on applications, we implemented a macrobenchmark using LogDB (Section 4.3). Using the benchmark, we compared Ziplog with Scalog, each with six shards. Each benchmark repeatedly performs the following transaction, using integers for both keys and values:

```
const int N = 10;
int k = rand()%N;
Txn t = newTxn();
int v = t.read(k);
t.write(k, v+1);
t.commit();
```

In the transaction, a client reads the value v of a random key k between 0 and 9, then updates the value of key k to $v + 1$. Each client runs in a closed loop. We increase the number of clients and measure the throughput in terms of the total number of committed transactions per second and the abort rate.

Figure 4.8 presents the throughput with a variable number of clients. It shows that LogDB on Ziplog achieves higher throughput than LogDB on Scalog. The difference is explained by the significantly reduced abort rate due to Ziplog's lower latency (see Figure 4.9).

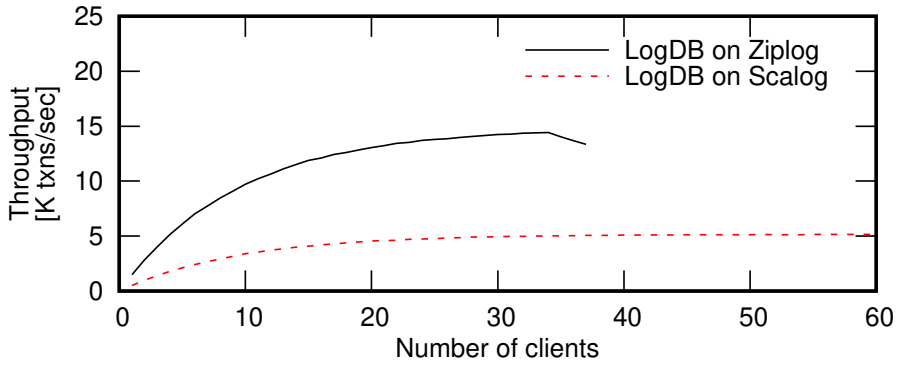


Figure 4.8: Throughput vs. number of clients in LogDB

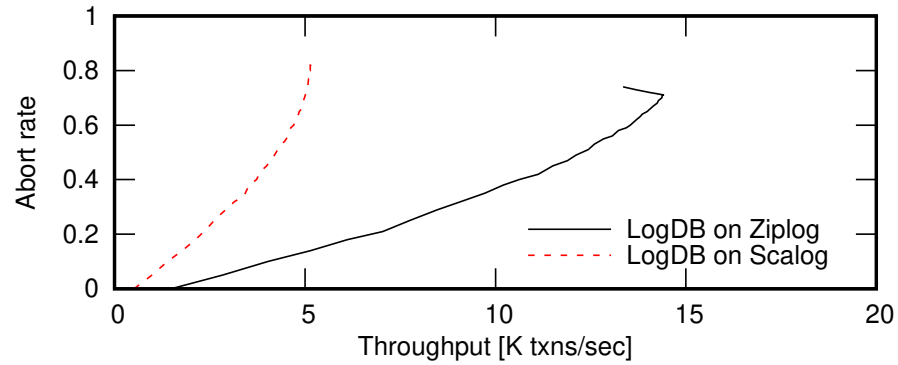


Figure 4.9: Abort rate vs. throughput in LogDB

4.5 Conclusion

To the best of our knowledge, Ziplog is the first "one-size-fits-all" shared log design, achieving all three of low latency, high throughput, and total order. One of the crucial insights is a novel linearizable `InsertAfter` API, offering different semantics than the `Append` interface existing shared logs provide while still supporting total order as well as the ordering of dependent records. Through this insight, Ziplog can support both sharding for scalable throughput and low latency. Even though Ziplog supports sharding and total order, clients only need to interact with a single shard in the normal case. Without the need for special hardware, Ziplog can add records in two message delays when there is no contention and in three message delays when there is. Compared to the

state-of-the-art, our evaluation shows that Ziplog achieves throughput comparable to Scalog and latency comparable to NOPaxos.

CHAPTER 5

CONCLUSION

This dissertation has explored the requirements of an ideal shared log implementation, analyzed the challenges of implementing such a shared log, and used two prototype systems, Scalog and Ziplog, to demonstrate how to achieve seamless reconfiguration, flexible data placement, and low latency in a totally ordered, scalable shared log.

When we ran experiments with Scalog and Ziplog, we found that seamless reconfiguration shines not only when the demands of applications change, but also when the shared log implementation is upgraded. During a typical upgrade process, servers are upgraded and restarted one by one, through a process analogous to what happens during failure recovery. When many servers need to be upgraded, the hiccups in availability caused by this process make the shared log service virtually unusable during the upgrade. With seamless reconfiguration, only clients connected to the restarting shard are affected, which reduces disruptions during the upgrade. Better, each Scalog shard can be configured with $2f+1$ storage servers (see the "mask" failure recovery option in Section 3.3.4), which allows restarting servers without triggering failure recovery.

The current implementations of Scalog and Ziplog only work within a datacenter. If Scalog were deployed across datacenters, clients would not be able to differentiate between storage servers within a shard partitioned from each other and the entire shard partitioned from the ordering layer. If Ziplog were deployed across datacenters, then network partition between the membership management module and a shard would prevent the shard from proceeding to the next epoch and stop sequential access to the log. How to implement

a geo-distributed, totally ordered, shared log remains an open problem. One potential direction to solve the problem is to weaken the consistency guarantee for records stored in shards belonging to different datacenters.

As discussed in Chapter 4, the current Ziplog implementation focuses on achieving a latency comparable to that of NOPaxos without using special hardware. However, it is possible to further reduce Ziplog's latency with state-of-the-art technology such as RDMA. The challenges are twofold. First, how can Ziplog best use the low latency and high throughput of RDMA? Second, given that RDMA allows one server to write data to another server's memory *without* notifying the log implementation, how can Ziplog enforce strong consistency while maintaining low latency? Although a complete discussion of an RDMA-version of Ziplog is beyond the scope of this dissertation, at the time of this dissertation's writing, we are working on an implementation of the RDMA-version of Ziplog.

The ordering-as-a-service paradigm is compelling, and shared logs provide an excellent abstraction for implementing such an ordering service. Although we have implemented two shared logs and explored the scope of applications that could benefit from a shared log, the shared log abstraction remains under-explored. There remain challenges to making the shared log abstraction usable to a broader set of applications.

First, the `subscribe` interface in Scalog and Ziplog requires log consumers to read the entire log; but with current technology, no single server can consume messages at the speed Scalog and Ziplog provide. A necessary step then would be to allow consumers only to receive records they are interested in. One may consider specifying the subset of records using a filter or a set of topics,

but doing so causes other issues. For example, the global sequence numbers corresponding to the records received by a consumer would no longer be guaranteed to be consecutive, so how can consumers detect gaps between records? We believe that more experience with real-world applications will be critical to gain greater insight into how to address this issue.

Second, instead of a powerful, single log supporting a high-performance database, applications may require or prefer many little logs to build millions of tiny databases [30]. Scalog and Ziplog are not designed to support a large number of little logs. The challenges include determining which physical servers should store which little logs and, given that one little log might be stored in only one shard, understanding how to maintain each little log's availability when a shard fails.

Finally, to provide an ordering service, it is necessary to support multi-tenancy, that is, having a single instance of the log on a cluster serving multiple customers. It is challenging to separate customers while maintaining high throughput and low latency. Following the concept of virtualization in other operating systems and computer networks, one may virtualize the log and have each customer be served by a virtual log.

BIBLIOGRAPHY

[1] Achieving high concurrency with ApsaraDB for RDS. https://www.alibabacloud.com/blog/achieving-high-concurrency-with-apsaradb-for-rds_594297.

[2] AlibabaMQ for Apache RocketMQ. https://www.alibabacloud.com/product/mq.

[3] Amazon Kinesis. https://aws.amazon.com/kinesis/.

[4] CloudLab. https://cloudlab.us.

[5] CorfuDB. https://github.com/CorfuDB/CorfuDB.

[6] Data Plane Development Kit. https://dpdk.org/.

[7] DistributedLog. http://bookkeeper.apache.org/distributedlog/.

[8] The Go programming language. https://golang.org.

[9] Google Cloud Pub/Sub. https://cloud.google.com/pubsub/.

[10] Google protocol buffers. https://developers.google.com/protocol-buffers/.

[11] IBM MQ. https://www.ibm.com/products/mq.

[12] Kafka uses. https://kafka.apache.org/uses.

[13] Kubernetes. http://kubernetes.org.

[14] LogDevice: distributed storage for sequential data. https://logdevice.io/.

[15] Microsoft Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/.

[16] Oracle Messaging Cloud Service. https://www.oracle.com/application-development/cloud-services/messaging/.

[17] Paxos variants. http://paxos.systems/variants.html.

[18] Remote Direct Memory Access. https://en.wikipedia.org/wiki/Remote_direct_memory_access.

[19] Storage Performance Development Kit. https://spdk.io/.

[20] Taobao. https://www.taobao.com.

[21] Why brands are fighting over milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/.

[22] Zlog: A high-performance distributed shared-log for Ceph. https://github.com/cruzdb/zlog.

[23] Daniel Abadi. Partitioned consensus and its impact on Spanner's latency. https://dbmsmusings.blogspot.com/2018/12/partitioned-consensus-and-its-impact-on.html, 2018.

[24] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[25] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Crash consistency: fsck and journaling. In *Operating Systems: Three Easy Pieces*. 2018.

[26] 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 conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[27] 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)*, 2013.

[28] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[29] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[30] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Proceedings of the 17th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[31] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-Backup approach. *Distributed systems*, 1993.

[32] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 1996.

[33] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Preceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[34] 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)*, Hollywood, CA, 2012. USENIX Association.

[35] Emma Dauterman and Zoë Bohn. Network-Ordered Paxos on a cloud platform. In *Reproducing Network Research, Course Project of CS 244, Stanford University*, 2018. `https://reproducingnetworkresearch.files.wordpress.com/2018/07/bohn_dautermann.pdf`.

[36] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3), 1985.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[38] 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)*, 2020.

[39] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[40] Felipe Dutra Tine e Silva. Kafka: ordering guarantees. `https://medium.com/@felipedutratine/kafka-ordering-guarantees-99320db8f87f`.

[41] Raul Castro Fernandez, Peter R Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. Liquid: Unifying nearline and offline big data integration. In *CIDR*, 2015.

[42] Colin Fidge. Fundamentals of distributed system observation. *Software*, 1996.

[43] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the International Conference on Foundations of Computations Theory, Lecture Notes in Computer Science*, volume 158. Springer, 1983.

[44] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

[45] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 2003.

[46] Gregory R Ganger and Yale N Patt. Metadata update performance in file systems. In *Proceedings of 1th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[47] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2), 2012.

[48] Rachid Guerraoui and Andre Schiper. Total order multicast to multiple

groups. In *Proceedings of 17th International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[49] Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 1984.

[50] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

[51] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[52] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 2019.

[53] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2000.

[54] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the VLDB Endowment*, 2000.

[55] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, 2007.

[56] Jay Kreps. The log: what every software engineer should know about real-time data's unifying abstraction. `https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`.

[57] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[58] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 1981.

[59] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

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

[62] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 2001.

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

[64] Leslie Lamport. Fast Paxos. *Distributed Computing*, 2006.

[65] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, 2009.

[66] Leslie Lamport and Mike Massa. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, 2004.

[67] 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 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[68] 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 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[69] Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. Stoppable paxos. Technical report, Technical Report MSR-TR-2008-192, Microsoft Research, 2008.

[70] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[71] 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)*, 2013.

[72] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4), 2016.

[73] 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 conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[74] Kun Ren, Alexander Thomson, and Daniel J Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. In *Proceedings of the VLDB Endowment*, 2014.

[75] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[76] Richard D Schlichting and Fred B Schneider. Fail-Stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3), 1983.

[77] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[78] Alexander Thomson and Daniel J Abadi. The case for determinism in database systems. In *Proceedings of the VLDB Endowment*, 2010.

[79] 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*, 2012.

[80] Robbert van Renesse. Paxos made moderately complex. *ACM Computing Surveys*, 2011.

[81] Robbert van Renesse and Fred B Schneider. Chain replication for support-

ing high throughput and availability. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[82] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with Apache Kafka. In *Proceedings of the VLDB Endowment*, 2015.

[83] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J Freedman, and Dahlia Malkhi. vCorfu: a cloud-scale object store on a shared log. In *Proceedings of 14th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2017.