Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log

Cong Ding, David Chu, and Evan Zhao, *Cornell University;* Xiang Li, *Alibaba Group;*
Lorenzo Alvisi and Robbert van Renesse, *Cornell University*

https://www.usenix.org/conference/nsdi20/presentation/ding

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the
17th USENIX Symposium on Networked
Systems Design and Implementation
(NSDI '20) is sponsored by

Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log

Cong Ding David Chu Evan Zhao Xiang Li[†] Lorenzo Alvisi Robbert van Renesse
Cornell University [†]*Alibaba Group*

Abstract

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. *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 reconfiguration with no loss in availability, and recovers quickly from failures. At the same time, Scalog provides high throughput and total order.

The paper describes the design and implementation of Scalog and presents examples of applications running upon it. To evaluate Scalog at scale, we use a combination of real experiments and emulation. Using 4KB records, a 10 Gbps infrastructure, and SSDs, Scalog can totally order up to 52 million records per second.

1 Introduction

A shared log[1] offers a simple and powerful abstraction: a sequence of ordered records that can be accessed and appended to by multiple clients. This combination of power and simplicity makes them a popular building block for many modern datacenter applications. All cloud providers offer a shared log service (e.g., AlibabaMQ [2], Amazon Kinesis [3], Google Pub/Sub [8], IBM MQ [10], Microsoft Event Hubs [13], and Oracle Messaging Cloud Service [14]), which is also available through multiple open source implementations (e.g, Apache Kafka [36], Corfu [21], and FuzzyLog [41]).

Shared logs are used to (1) record and analyze web accesses for recommendations, ad placement, intrusion detection, performance debugging, etc. [11, 31, 36, 50]; (2) prepare a transport between stages in a processing pipeline that may be replayed for failure recovery [11, 50], and more broadly; (3) address the trade-off between scalability and consistency [51]. Consider, for instance, deterministic databases [22, 34, 35, 47, 48]: retrieving transactions from a single shared log allows these databases to shorten or eliminate distributed commit protocols, avoid distributed deadlocks, and achieve, in principle, superior transactional scalability [44, 47, 48].

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; and scale write throughput without giving up on total order. 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.

The state-of-the-art Corfu [21] 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 has been committed at all storage servers and clients. The Corfu data layout is defined by an inflexible round-robin policy, with significant performance implications: for example, reads require playing back the log where relevant updates are interspersed with unrelated records (a potential performance bottleneck) and writes cannot be directed to the closest storage server to reduce latency. vCorfu [51], an object store based on Corfu, addresses the issue of slow reads by complementing the Corfu shared log with *materialized streams*, log-like data structures that store together updates that refer to the same object. 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 [25]—vCorfu is at risk of losing data. Finally, 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 [22] achieves, to the best of our knowledge, the best throughput among today's totally ordered shared logs but, at about 570K writes/sec, its performance falls short of the needs of applications like Taobao [16], Alibaba's online market, which ran millions of database writes/sec at its 2017 peak [1].

[1] A shared log is also known as a *message bus*, but not all message buses provide a durable message store.

Scalog, the new shared log that this paper introduces, aims to address these limitations. 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 [21,22,51]: it assumes that faulty servers will *crash* [29], rather than *fail-stop* [45], thus sidestepping the so-called "split-brain syndrome" [26].[2] 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 responsible 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 [12]), 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 [36] 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 [18, 27, 28, 44]. 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:

- It provides seamless reconfiguration. Our prototype sees no increase in latency or drop in throughput while Scalog is being reconfigured.
- It 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.
- It offers (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: its maximum throughput is essentially limited by the maximum number of shards times the maximum throughput of each shard. With 17 shards, each with two storage servers, each processing 15K writes/sec, our prototype achieves a total throughput of 255K totally ordered writes/sec. Through emulation, we demonstrate that, at a latency of about 1.6 ms, Scalog can handle about 3,500 shards, or about 52M writes/sec—two orders of magnitude higher than the best reported value to date at comparable latency [22].

2 Motivation and Design

We motivate Scalog and its design principles with an online marketplace that enables sellers to list and advertise their merchandise, and allows buyers to browse and purchase. Page views and purchases are stored in a log that is used for multiple purposes, including extracting statistics (such as the number of unique visitors), training machine learning algorithms that can recommend merchandise and sellers to buyers, and simplifying fault tolerance by providing applications with a shared "ground truth" about the system's state.

This example highlights six key requirements for the underlying shared log. First, the log requires *auto-scaling*, the

[2] 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.

`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 1: Scalog API

ability to dynamically increase or decrease available throughput as needs change (e.g., during the holiday seasons) without causing any downtime. Second, for reproducibility during debugging and consistent failure recovery, the log should be totally ordered. Third, the log must minimize latency, for example, by allowing log clients to write to the nearest replica. Fourth, the log must provide high append throughput to support large volumes of store activity. Fifth, the log must provide high sequential read throughput to support analytics, which periodically read sub-sequences of the log. Finally, the log must be fault tolerant, as the online services that depend on it should be uninterrupted. Below, we discuss how Scalog addresses these requirements.

2.1 Scalog API

Scalog provides the abstraction of a totally ordered shared log. Table 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 committed 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.[3] Note that these guarantees are sufficient to implement a replicated state machine [46] 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 [32]—the strongest possible consistency guarantee.

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 §3). 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.

2.2 "Order first" Considered Harmful

Current totally ordered log implementations [6, 12, 21, 22] share a similar architecture: to append a new record, a client first obtains the record's position in the log, the *log sequence number*, via some *sequencer*, and then proceeds to make the record persistent. This design raises two 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 [20,30]. 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 ex-

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

pensive; 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 [21, 22].

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 [40]) or write-once disks (e.g., Corfu [21]).

2.3 Scalog Design Overview

By adopting a persistence-first architecture, Scalog avoids these 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 [50]; 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. Concurrently, the affected shard is finalized.

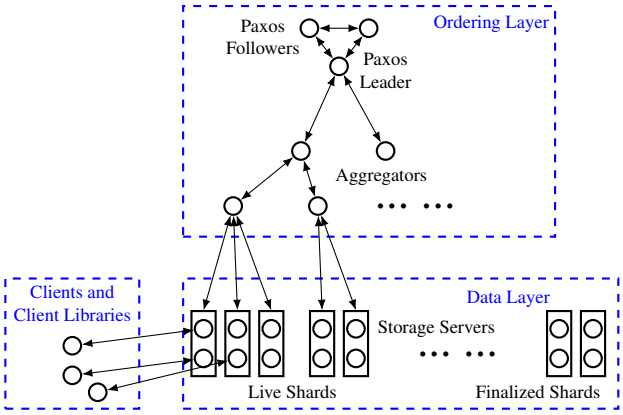

Figure 1: Scalog's architecture: arrows denote communication links; circles denote servers; each rectangle denotes one shard. Servers in the same shard communicate with each other. In this example, both shards and the Paxos instance in the ordering layer are configured to tolerate one crash.

3 Architecture

Figure 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.

Client Library. The library implements the Scalog API and communicates with the data layer (see Table 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 [38]. The second concern is that the over-

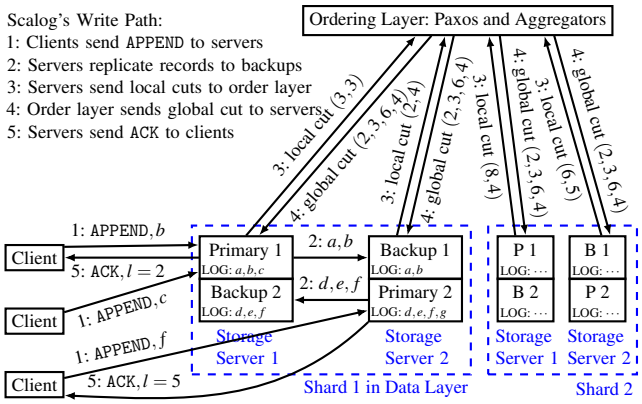

Figure 2: Scalog message flow for `append` operations

head 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 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 state 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.

4 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.

4.1 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 2, Step 1) and awaits an acknowledgment.

As shown in Step 2 of Figure 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) [19, 24]: 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 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 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 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 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.

4.2 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 §4.1 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 §4.3), 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).

4.3 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.

4.4 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.

4.4.1 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 §4.4.2); these failed shards are finalized immediately.

4.4.2 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 §6.2).

Applications that require data locality may run application processes in storage servers (see §5.3). *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 [25].

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.

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

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 2 summarizes these trade-offs and compares these options with failure recovery in Corfu and vCorfu.

4.4.3 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 [17, 21]), throughput goes back to normal instead of compensating for the loss of availability.

5 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.

5.1 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 §2, 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 §2, 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.

5.2 Scalog-Store

Modeled after Corfu-Store [21], 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.

5.3 vScalog

Modeled after vCorfu [51], 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.

6 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? (§6.1)
- How well does Scalog handle failures? (§6.2)
- How much write throughput can Scalog achieve and what is the latency of its write operations? (§6.3)
- How well do Scalog read operations perform in different settings? (§6.4)
- How do Scalog applications perform? (§6.5)

We have implemented a prototype of Scalog in golang [7], using Google protocol buffers [9] 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, 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.

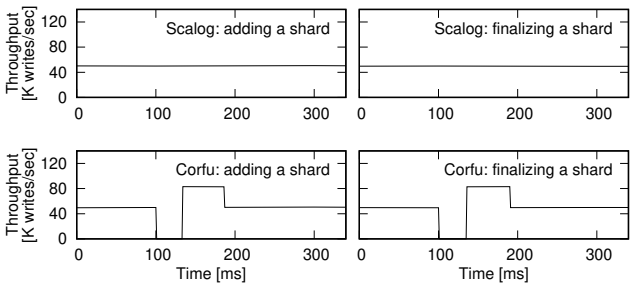

Figure 3: Throughput during reconfiguration

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 5 in §6.3.2).

6.1 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 setting. We either add a shard or finalize a shard at $t=100$ms.

Figure 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 §4.4 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 [21].

6.2 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 $t=2$s and measure how throughput is affected. Figure 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 com-

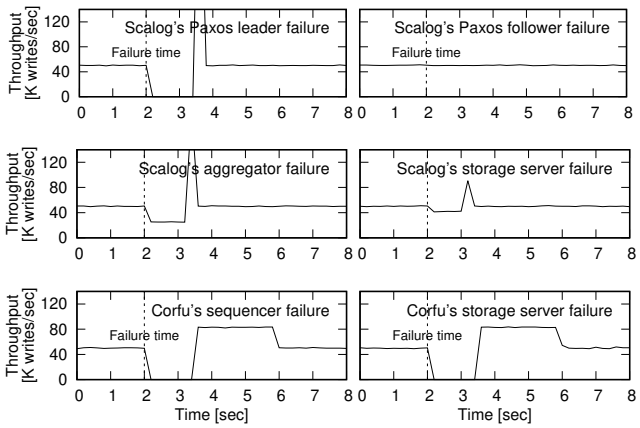

Figure 4: Throughput under different failure scenarios

mitted 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 unavailable [21]. Heavy contention among clients causes Corfu's throughput to drop to nearly zero until a replacement sequencer joins [17]. 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 [21]. Again, once the failure recovery completes, Corfu is saturated until all buffered records are stored.

6.3 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 §6.3.2, even with this short interval, Scalog already supports many more storage servers than we have resources to deploy.

6.3.1 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 [21], 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 [22].

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,

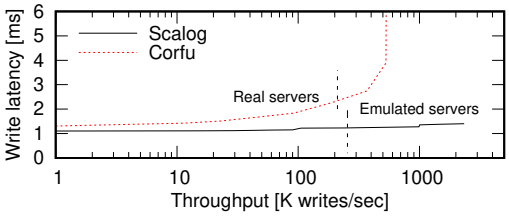

Figure 5: Latency vs throughput for Scalog and Corfu. The vertical dotted line separates results obtained with real servers from those obtained through emulation. For Scalog, we emulate storage servers, but not aggregators. Scalog's maximum throughput in this configuration is limited by the number of machines available to us.

we never assign to the ordering layer or to individual aggregators more than half of 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.

6.3.2 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 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.

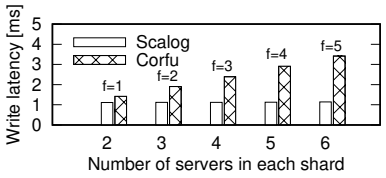

Figure 6: Write latency vs. shard size

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 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.[4] This throughput could be further increased by using a larger interleaving interval, trading latency for throughput.

Scalog's latency in Figure 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 [49] 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 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.

[4] 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.

6.4 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 §4.2). 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) (§4.2); 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 records/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.

6.5 Impact on Applications

We focus on the applications discussed in §5. 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.

6.5.1 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.

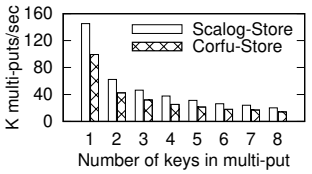

Figure 7: throughput of `multi-put` with 10 shards

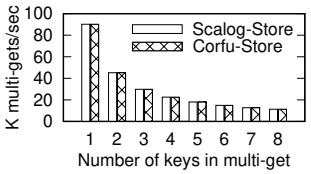

Figure 8: throughput of `multi-get` with 10 shards

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

Figure 7 shows that Scalog-Store has higher `multi-put` throughput than Corfu-Store, because each storage server in Scalog has higher throughput (see §6.3.1). 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 8) is limited by random read throughput of storage servers.

6.5.2 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 §6.3.1. 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 [51]).

7 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 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.

8 Related Work

The shared log abstraction is, implicitly or explicitly, at the core of state machine replication protocols [46], and Scalog draws inspiration from several of them.

In Vertical Paxos [39], configurations can change from slot to slot, allowing for seamless reconfiguration similar to Scalog. Like Scalog, EPaxos [43] allows all replicas to accept client requests. However, EPaxos only builds a *partial* order consistent with specified dependencies among records; in addition, maintaining and checking dependencies creates a bottleneck. In networks that almost never reorder messages, NOPaxos [40] achieves very high throughput and low latency using a custom hardware switch to order records.

Mencius [42] and Derecho [23] partition log slots among multiple leaders. Essentially, each process creates a locally ordered log, which is then interleaved in round-robin order. Similarly, Calvin [48] dispatches to multiple sequencers transaction requests, which are compiled into batches. The batches are then interleaved in round-robin order to build a total order. Scalog generalizes this idea and allows for more flexibility in how the logs are interleaved, which is not sensitive to slow servers.

Kafka [36], a widely-used shared log system, uses sharding to scale and provides total order within each shard, but not across them. Pravega [15] provides a sharded log similar to Kafka and focuses on a rich set of reconfiguration operations that support scaling. FuzzyLog [41] builds a partially ordered log by tracking Lamport's happened-before relation [37] between records stored in different shards. DistributedLog [6] also supports sharding and provides a totally ordered log, but its single-writer-multiple-reader access model is not conducive to high write throughput.

To provide both total order and high throughput, it is necessary to separate ordering from data dissemination. Like Scalog, Corfu [21] separates ordering from data dissemination and relies on sharding. A function, maintained in ZooKeeper [33], maps sequence numbers to shards. A client first obtains a sequence number for a record from the Corfu sequencer, and 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 [49].

LogDevice [12] is similar to Corfu, but replaces the mapping function with a non-deterministic record placement strategy. By allowing clients to write to any shard, LogDevice achieves flexible data placement. However, all records still need to be ordered by a sequencer similar to DistributedLog's single writer [6], limiting throughput.

9 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 essential 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 order of magnitude higher throughput than the state-of-art shared log implementation.

Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their insightful comments. We also thank Matthew Burke, Natacha Crooks, Youer Pu, Chunzhi Su, Florian Suri-Payer, Cheng Wang, and Yunhao Zhang for their valuable feedback on earlier drafts of this paper. This work was supported by NIST F568386 and by NSF grants CSR-1409555 and CNS-1704742.

References

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

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

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

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

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

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

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

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

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

[15] Pravega. http://pravega.io.

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

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

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

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

[20] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Crash consistency: fsck and journaling. In *Operating Systems: Three Easy Pieces*. 2018.

[21] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. 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.

[22] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[23] J. Behrens, K. Birman, S. Jha, M. Milano, E. Tremel, E. Bagdasaryan, T. Gkountouvas, W. Song, and R. van Renesse. Derecho: group communication at the speed of light. Technical report, Cornell University, 2016.

[24] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 1993.

[25] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: reducing the frequency of data loss in cloud storage. In *Preceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[26] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3), 1985.

[27] F. D. T. e Silva. Kafka: ordering guarantees. https://medium.com/@felipedutratine/kafka-ordering-guarantees-99320db8f87f, 2018.

[28] R. C. Fernandez, P. R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang. Liquid: unifying nearline and offline big data integration. In *CIDR*, 2015.

[29] M. 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.

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

[31] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building LinkedIn's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2), 2012.

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

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

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

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

[36] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

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

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

[40] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. 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.

[41] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The FuzzyLog: a partially ordered shared log. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[42] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[43] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[44] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. In *Proceedings of the VLDB Endowment*, 2014.

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

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

[47] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *Proceedings of the VLDB Endowment*, 2010.

[48] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[49] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[50] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. Building a replicated logging system with Apache Kafka. In *Proceedings of the VLDB Endowment*, 2015.

[51] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. 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.