Ziplog: A Totally Ordered Log combining Low Latency with Scalable Throughput

Cong Ding, Chaska Yamane, Evan Patrick, Lorenzo Alvisi, Robbert van Renesse

Cornell University

Abstract

The shared log abstraction has proved to be an important building block for distributed systems. However, none of the existing implementations achieve all three of low latency, high throughput, and total order. *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.

1 Introduction

Shared logs have emerged as a key building block for managing the complexity of building datacenter applications [1, 2, 6, 7, 11, 34]. 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 [14]) and in the background (e.g., to support debugging, analytics, intrusion detection, and failure recovery [8, 25, 34, 49]).

Much recent work [10, 14, 22, 50] has 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 [22], a state-of-the-art shared log implementation, reports a 1.2 ms latency for appending a record at the end of its log; in comparison, consensus decisions in NOPaxos [40], 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 paper, 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 [22]), total order and low latency for a single shard (as in NOPaxos [40]), or low latency and scalable throughput, but with no cross-shard total order (as in Kafka [34]).

Ziplog, the shared log that we present in this paper, 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* [27]. 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 [14] or Scalog's ordering layer [22]).

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 [37] 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 [22]: 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.

- It proposes a new `InsertAfter` interface to add new records to the log. While guaranteeing linearizability, this interface enables individual shard to make consistent decisions about the global order of records they store, avoiding cross-shard coordination in the common case and allowing Ziplog to achieve low write latency.
- It implements 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. Our evaluation shows that contention is rare, and more than 70% of operations are completed in two message delays, allowing for a latency similar to NOPaxos but without the use of special hardware.
- Its 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 to achieve a throughput comparable to Scalog's.
- Meanwhile, it provides seamless reconfiguration, without compromising total order or linearizability, again without using special hardware.

2 Background and Motivation

Modern applications use a multi-tiered architecture to ease development and maintenance. The current trend in cloud computing and serverless computing is to structure the system's architecture in two tiers: an application tier, which communicates with users and remains stateless, and a storage tier, where the state is held. This structure allows the application tier to scale horizontally by adding more application servers. Building a scalable storage tier remains challenging.

Traditionally, the storage tier is a relational database, such as MySQL and Oracle. Though these systems work well for many applications, their scalability falls short of today's most demanding applications [22, 50]. One way to bridge this gap is to trade consistency for scalability, as in NoSQL databases like Cassandra [36], HBase [48], and MongoDB [12], which delegate the enforcement of strong consistency guarantees to the application logic. However, this tradeoff increases the complexity of applications and requires application developers to understand how to deal with inconsistencies.

The shared log abstraction provides an opportunity to address the tension between scalability and consistency: recent systems like vCorfu [50] and vScalog [22] show that it is possible to achieve high throughput in a storage system while guaranteeing strong consistency.

2.1 The Shared Log Abstraction

The shared log abstraction provides three basic methods. 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.

A shared log implementation ideally achieves the following properties to satisfy the need of applications:

- *Total order*: A total order of records makes it easier for applications to maintain consistency.
- *Scalable throughput*: 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.
- *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.

No shared log abstraction today achieves all of these properties. For example, sharded totally ordered shared logs like Corfu and Scalog provide scalable throughput, but adding a new record to the log takes over 1 ms [14, 22]. NOPaxos, 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.

The latency figures of today's scalable shared logs can have serious negative implications for some of the applications that rely on them. Consider, for example, deterministic transactional databases [15, 30, 32, 44, 45] 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 [16], longer transactions thus can result in longer blocking and higher chances for deadlock; with optimistic concurrency control (OCC) [35], they can cause higher abort rates and reduce transactional throughput [26].

2.2 Ziplog: 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 [13] and DPDK [4], 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 paper. 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.

2.3 Reassessing the Shared Log API

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* [27].

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 inherent 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 [14], which uses a sequencer, this additional roundtrip happens *before* a record is stored, while in Scalog [22], 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 [38], 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, as we will see shortly.

2.4 Ziplog's Insight

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 Section 5, in most cases they cause no hiccups in performance.

Though the idea at the core of Ziplog's design is in principle straightforward, realizing it in practice raises two technical challenges, both stemming from Ziplog's ability to fill log positions out of order.

Failure handling. If a storage server in a shard fails, certain log positions may no longer be filled and leave holes in the global order; further, a failure may catch a record mid-way in the process of being replicated within a shard, leaving the log in an uncertain state. Ziplog uses a Paxos-based failure handling service to enable consistent failure recovery.

Changing workloads. Different shards may receive records from clients at different rates; furthermore, these rates may change over time. Uneven and dynamically changing workloads will fill log positions out of order, increasing query latency for subscribe operations, which must wait whenever they encounter a hole in the log. To address this issue, Ziplog's membership management service periodically assigns write rates to each shard based on historical information. The shards use these rates to deterministically compute global indexes from local indexes. If a shard does not receive enough records, it fills the remaining log positions with no-ops.

3 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 1). Application servers may co-locate with storage servers, depending on resource management policies and application requirements.

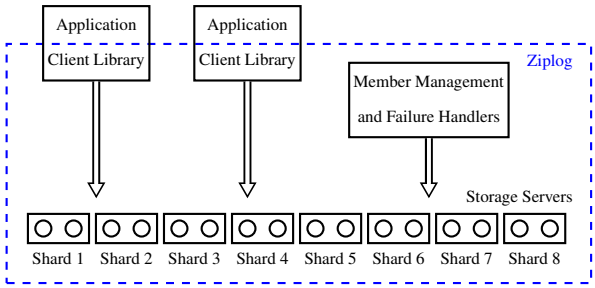


Figure 1: Ziplog's architecture

`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 1: Ziplog API

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 [20, 24]. 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 3.5).

3.1 Ziplog API

Table 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 [50] 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.

3.2 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 3.3) 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 3.4) 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 3.5) 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 $\langle$`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.

3.3 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 Section 2.4. 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 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

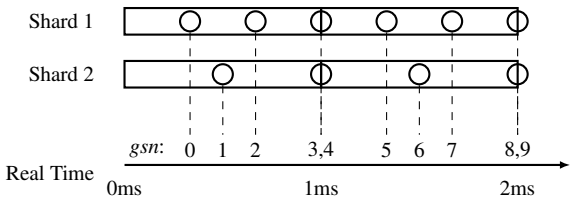


Figure 2: Example of *gsn* assignment. Record arrival rates at Shard 1 and Shard 2 are 3 records/ms and 2 records/ms, respectively. Circles, representing the expected arrival time of each record in each shard, are projected onto the real time axis to produce a total order.

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.

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 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 form its next broadcast. The shard will henceforth no longer accept new records, but will continue to be available for read operations.

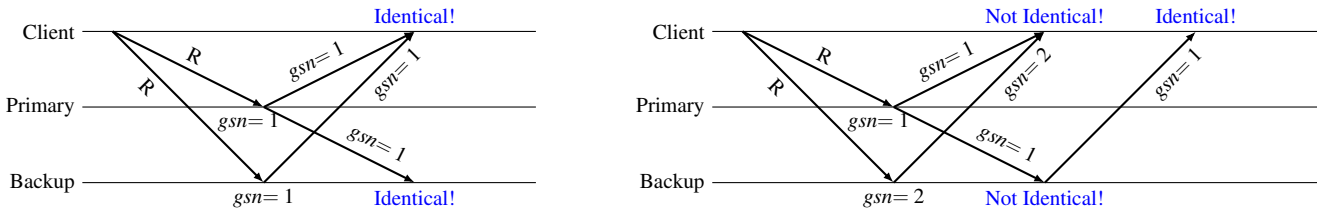
3.4 Replication

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 [40], 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 [21]. 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 3.3), (*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



(a) The client receives identical *gsn*s

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

Figure 3: Ziplog's Replication Protocol without and with contention

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 3a).

What if it doesn't? Then, the client continues waiting (Figure 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 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 [23]. 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 3.5 to ensure that all reads eventually complete.

3.5 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 3.4); (*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 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) [35].

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 [42].

5 Evaluation

Ziplog seeks to achieve total order, scalable throughput, and low latency. As discussed in Section 1, 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 [40], while for the latter it is Scalog [22].

In particular, we consider the following questions:

- Can Ziplog achieve low latency, competitive with NOPaxos? (Section 5.1)
- Can Ziplog achieve scalable throughput, competitive with Scalog? (Section 5.2)
- What is the performance of read operations in Ziplog? (Section 5.3)
- What is the impact of using the `InsertAfter` semantics? Does it affect Ziplog's subscribe operations under dynamic workloads? (Section 5.4)
- How do reconfigurations and failures impact Ziplog? (Section 5.5)
- How do Ziplog applications perform? (Section 5.6)

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 [21], the sequencer in our NOPaxos implementation is a server (i.e., NOPaxos's *end-host sequencing configuration* [40]) and uses multiple TCP connections to broadcast messages. Because Ziplog writes records to disks asynchronously (Section 3.4), our Scalog and NOPaxos implementations do the same. We used 4KB records, consistent with published results for Scalog [22] and Corfu [14].

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 [3]. 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 [22]).

5.1 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 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

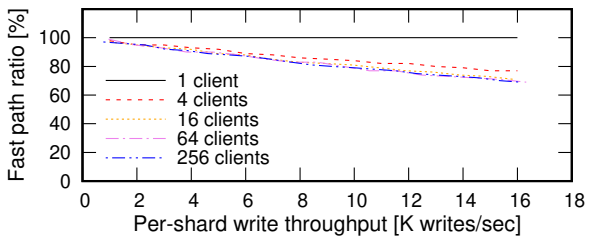


Figure 4: Ziplog's fast path ratio vs. per-shard throughput

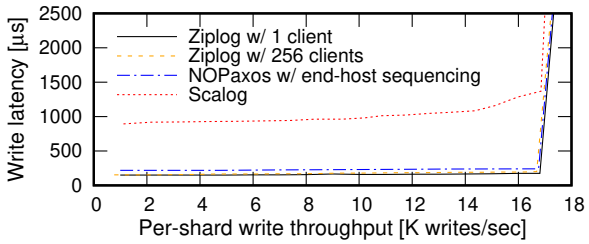


Figure 5: Average write latency vs. per-shard throughput

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 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 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 [40]). Scalog achieves 900 μs latency, which is better than the figure published in the Scalog paper [22] on the same hardware because we modified the Scalog code to write to disks asynchronously.

5.2 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 5 shows that a single shard can achieve approximately 16.8K writes/sec. To determine the maximum number of shards Ziplog can support, 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 broadcast 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 [22]. The broadcast interval does not affect write latency; however, for reasons discussed in Section 3.3, it does affect sequential access latency (Section 5.3).

5.3 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, 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 3.4)—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.

5.4 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

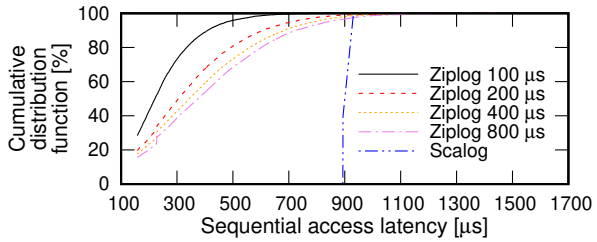


Figure 6: Sequential access latency with dynamic workloads for varying average duration between operations in each client compared to Scalog.

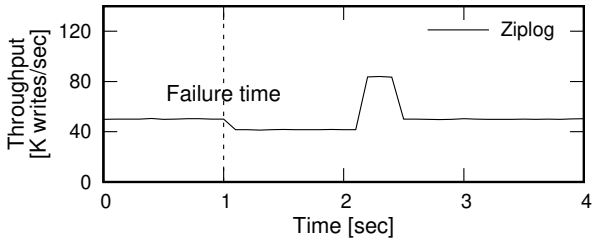


Figure 7: Throughput during failure recovery

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 6 presents the distribution of sequential access latency for varying T. As expected, a 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.

5.5 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 [22].

As discussed in Section 3.3, 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 $t = 1$s, we simulated a failure by killing one of the storage servers. Figure 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.

5.6 Impact on LogDB

To understand the effect of lower latency on applications, we implemented a macrobenchmark using LogDB (Section 4). 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 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 9).

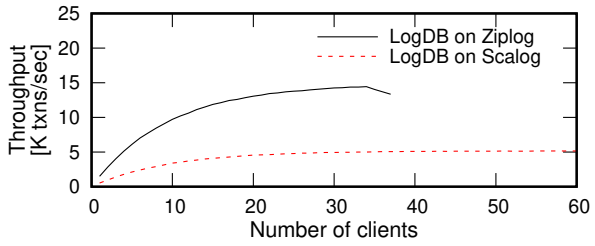


Figure 8: Throughput vs. number of clients in LogDB

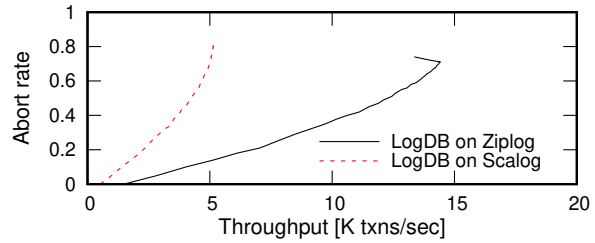


Figure 9: Abort rate vs. throughput in LogDB

6 Related Work

Several systems, including Zyzzyva [33] and Speculative Paxos [41] have demonstrated achieving total order in two message delays under favorable circumstances. NOPaxos [40] achieves total order in two message delays using special hardware. None of these systems are sharded and therefore have limited throughput. A major contribution of Ziplog is demonstrating that it is possible to achieve both low latency and scalable throughput in a totally ordered log. Moreover, Ziplog achieves this without using special hardware.

Ziplog's design builds on existing sharded log implementations to achieve scalable throughput. Kafka [34], deployed widely in industry, partitions the log into a collection of shards to scale out, but does not provide total order across shards. Corfu [14] provides a global total order, and uses a sequencer to avoid having to interact with multiple shards for each request. Communicating with the sequencer involves two message delays; further, shards make records persistent using chain replication [47], which adds $f+2$ additional message delays for a total of $f+4$ message delays in the best case.

Scalog [22] uses an ordering service that collects information from all the storage servers and then makes ordering decisions in batches across all shards in timed intervals. This way, Scalog can achieve high throughput, while latency is determined by the length of the interval, which is on the order of a millisecond or more. Ziplog's novel design uses no ordering services and yet scales to more shards than Scalog; further, removing the ordering service from the critical path allows Ziplog to lower latency.

Derecho [29] is a sharded and persistent group communication facility, built over an RDMA infrastructure. The best-case end-to-end latency is four message latencies, which is twice that of Ziplog. 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. Derecho supports per-shard total ordering (but global agreement on membership). Ziplog can support a large number of shards, resulting in a high aggregate throughput, while maintaining total order.

State machine replication protocols essentially build a shared log, though, lacking log partitioning, they do not provide high throughput. Primary/Backup [17], working in a fail-stop model [43], uses a primary server to sequence and broadcast records to backups, while Paxos [19, 39, 46] similarly replicates a log, but works in the weaker crash failure model that Ziplog also adopts. Ziplog's "slow path" within a shard resembles a Primary/Backup protocol and achieves a similar latency. However, by ordering records speculatively in its "fast path", Ziplog achieve a lower overall latency than Primary/Backup and Paxos; at the same time, Ziplog can provide higher throughput by partitioning the log.

There are several existing membership management and lock systems, similar to Ziplog's membership management service. Zookeeper [28] uses Zab [31], an atomic broadcast protocol, to implement a coordination service. Zookeeper serves as Corfu's membership management and mapping function assignment, while Ziplog implements its own membership management for its particular demands. Similarly, Chubby [18] implements a distributed lock service to provide coarse-grained locking and low-volume storage for loosely-coupled distributed systems. Kubernetes [9], Google's open-source resource management system, not only uses etcd [5] to manage membership information but also automatically allocates resources and deploys containers, which could help Ziplog to find the most suitable servers to add new shards and to replace failed servers.

7 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 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.

References

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

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

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

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

[5] etcd: a distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

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

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

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

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

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

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

[12] MongoDB: the most popular database for modern apps. https://www.mongodb.com/.

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

[14] 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.

[15] 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.

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

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

[18] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[19] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: high-performance broadcast for primary-backup systems. In *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011.

[32] 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.

[33] 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.

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

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

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

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

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

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

[48] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, 2011.

[49] 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.

[50] 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.