Supporting Streams in Shared Logs

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Rafael Sargento — 103344*
rafael.sargento@tecnico.ulisboa.pt

Advisors: Professor Luís Rodrigues

Abstract Shared logs are a key component of modern distributed systems. Typically, shared logs keep a persistent, totally ordered sequence of records that can be used to coordinate multiple servers running the same or different applications. For instance, if records store deterministic commands, replicas of an application may use the log to execute the same set of commands, in the same order, to implement state-machine replication. If the log is shared by many distinct applications, some servers may be interested in reading just a subset (denoted a *stream*) of the log entries. This requires efficient mechanisms that allow servers to skip entries that are not of interest to them. This work surveys the main techniques that have been proposed to provide stream support in distributed shared logs and proposes a design that aims at improving existing work.

Keywords — Distributed Systems, Shared Logs, Streams, Materialized Views

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

Shared logs are a key component of modern distributed systems. They keep a persistent, totally ordered, sequence of records that can be leveraged to coordinate multiple servers running the same or different applications. The literature is rich in shared log-based applications, including State-Machine Replication [1], Object Stores [2, 3], FaaS runtime engines [4] and Lock Systems [5, 6].

Although the shared log provides a unified global ordering of events, not all services may be interested in all records. For example, in a database system that supports cross-partition transactions, each partition is only interested in records that access their key-set [2, 3]. In addition, some systems leverage the same log infrastructure to support multiple independent applications: Boki [4] uses the same physical log to support a fault-tolerant FaaS workflow framework, an object store, and a message queue.

An application can always read all log entries and discard those that are not of interest to it. Unfortunately, this approach is inefficient. First, even if the application has quick access to the log entries, parsing the log to find the relevant entries is time-consuming. Second, most logs are distributed and replicate log entries so that they can be read with low latency. Replicating a log entry on a server whose content is irrelevant to its application is a waste of resources. Due to this problem, several log designs [2–4, 7] support *streams* (sometimes, also called materialized *views*) which enable applications to efficiently read only the log entries in which they are interested.

This work surveys the main techniques that have been proposed to support streams in distributed shared logs and proposes a design that aims at improving existing work. The remainder of this document is structured as follows. Section 2 identifies the goals of our work and its expected contributions. Section 3 provides the relevant background and surveys the related work. Section 4 describes the plan to improve on the state-of-the-art and Section 5 describes the proposed evaluation techniques. Section 6 provides a schedule for future work and Section 7 concludes this report.

2 Goals and Expected Results

This work addresses the problem of building a distributed shared log for scenarios where different clients are interested in different subsets of the log entries. More precisely:

> *Goals:* Extend existing shared logs with techniques that allow clients to consume only a subset of log entries efficiently, in particular they should prevent a client from blocking while waiting for a record it is not interested in.

The proposed design will leverage ideas from ZipLog and Hydra [8] to achieve a shared log implementation that enables clients to efficiently track the latest appended entries of interest in the log. The project will produce the following expected results:

> *Expected results:* A survey of related work. The proposal of a novel algorithm to support multiple streams in a shared log. An implementation and evaluation of the proposed algorithm.

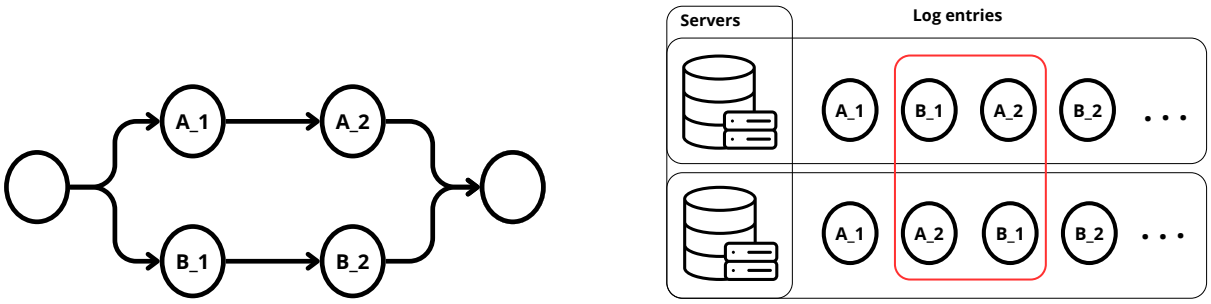
3 Background and Related Work

This section provides the necessary background on shared log abstractions and surveys relevant system implementations. The observations and limitations identified here motivate the design of the proposed system, which is described in Section 4.

3.1 The Shared Log Abstraction

Shared logs allow multiple servers to store and read ordered records of data, i.e. log entries. These log records are persistently stored. The shared log abstraction provides an append-only interface, which means that records are immutable, and their binding to a specific log position is definitive; once appended, the position of a record is permanent and will always refer to that same record. Servers may only add new records or read existing ones.

Shared logs also provide ordering guarantees. Some implementations provide total order, where all servers observe the same global sequence of records, while others offer partial order. Figure 1 illustrates an example of partial ordering, where two servers receive operations from clients *A* and *B*. Although both servers observe an order of operations that preserves per client ordering, the global order of operations differs between servers. In contrast, in a totally ordered shared log, all servers observe the same global order of operations.



(a) Dependency constraints in partial ordering: for each client, the first operation must be ordered before the second

(b) Example of a partial order in which operations without dependencies may be ordered differently across servers

Figure 1: Partial ordering example

In terms of performance, shared logs must scale to support high-throughput workloads, handling concurrent operations from multiple clients without jeopardizing the properties introduced above. A common way to scale the system is to partition the log across multiple servers (often denoted as *shards*) that store and serve a subset of the log entries.

Distributed shared logs are typically implemented by multiple processes, each with a different role. Common roles include shard servers, sequencers, and clients. The shard servers are responsible for storing and serving a subset of all log entries. Sequencers are used to establish the ordering of records. Finally, clients can either append new records or read existing ones. Additionally, clients can also become subscribers to the log, to receive new log entries as they are appended.

Clients that read log entries process each appended record to extract the contained information, and subsequently can report this information to relevant applications or services. This procedure is commonly referred to as the delivery of the operation.

The common exposed API for shared logs typically includes three main operations: `append`, `read`, and `subscribe`, as summarized in Table 1.

Operation	Description
`append`	Adds a new record to the log.
`read`	Retrieves a specific log entry.
`subscribe`	Allows clients to continuously track newly appended records.

Table 1: Common API operations for shared logs

3.2 Relevant Shared Logs

This section presents a brief survey of some of the most relevant shared log designs that can be found in the literature.

3.2.1 Corfu

Corfu [9] implements a totally ordered shared log, with linearizable [10] appends, which aims at achieving high throughput by letting different log entries be written in parallel to different storage units. Corfu uses multiple Solid-State Drives (SSD) with write-once semantics to store log entries. The SSD drives are organized into

replication groups, each responsible for storing and serving a different portion of the log. The mapping between log entries and replication groups, called a *projection*, is known by all participants.

To perform an append, clients must first determine the log's tail, i.e. its next free position. In Corfu, clients can achieve this using two different complementary techniques. First, the client can read the log entries sequentially until it finds an empty entry. Then it attempts to write on that entry. If two clients attempt to write on the same entry concurrently, the write-once semantics of the SSD units ensures that only one of them succeeds. If the write fails, the client will continue to look for the next empty log position and repeat this process. Alternatively, the client can find the next free entry with the help of a centralized sequencer process. Instead of iterating through the log, the clients first contact the sequencer, which provides the next empty log position. This is faster and prevents different clients from performing concurrent write attempts at the same log position. The latter approach is used by default, but a client can always fall back to sequential parsing if the sequencer becomes unresponsive.

As noted above, each entry is replicated in multiple storage units. To ensure the consistency of replicas, Corfu uses a variant of Chain Replication [11], where the client updates all replicas in sequence, starting from the head until it reaches the tail. Note that if two clients attempt to write to the same log position concurrently, only one will succeed in writing to the head of the chain, and therefore only one client will be allowed to update the remaining replicas. If a client fails during this process, a special recovery procedure is invoked. The recovery procedure also handles the case where a client obtains an entry from the sequencer but never updates any of the replicas, marking the entry as a "no-op".

Corfu provides high scalability by supporting parallel append operations. However, its design still presents significant challenges. First, while the sequencer improves contention on append, the sequencer itself becomes a contention point and can become a bottleneck to system performance. Second, Corfu requires all nodes to agree on the mapping between entries and storage units, which introduces a significant coordination overhead during reconfiguration. Third, in Corfu, the log position is assigned before data is persisted, which causes "holes" in the log, when clients crash during the write operation. This prevents progress in reading the log, as clients must wait for the recovery procedure to decide if they can safely ignore this position or should wait for an incoming record.

3.2.2 Scalog

Scalog [12] is an alternative shared log design that, like Corfu, offers total order. In contrast to Corfu, Scalog first stores log entries and then assigns a global order to operations. This approach allows Scalog to avoid blocking readers if a client crashes during a write operation, enabling seamless reconfiguration.

The architecture of Scalog is split into the data layer and the ordering layer. Similarly to Corfu, the storage nodes in the data layer receive client requests. Scalog servers are also divided into multiple fault-tolerant groups, called shards, which are responsible for storing and replicating data using a variant of the primary-backup protocol [13, 14].

This system uses an ordering layer responsible for assigning global sequence numbers to log entries. The ordering layer is implemented using Paxos [15] for fault tolerance. Scalog also uses a middleware layer of *aggregators* to connect the ordering layer and the data layer. The aggregators are used to reduce the load on the machines ordering records, aggregating requests from multiple storage servers.

To execute an append operation, clients first forward the record to a storage server, which replicates the new data to the backup servers, ensuring fault-tolerance. The storage server periodically reports newly stored log entries to the ordering layer. The ordering layer aggregates these reports and informs all servers in the data layer of which records have been successfully replicated. Using this information, the data layer induces the global order of operations via a deterministic ordering function.

One advantage of Scalog over Corfu is that entries are only submitted to the log after being replicated. Thus, any delays in the replication procedure do not affect the append/ read latency of other entries. Furthermore, the use of an aggregation layer also mitigates eventual bottlenecks at the sequencer nodes. The main disadvantage of Scalog is the latency penalty in the ordering service introduced by the batching scheme, given that a request must wait for a batch to complete at multiple levels of the aggregation topology.

3.2.3 NOPaxos

NOPaxos (Network Ordered Paxos) [16] is a protocol that leverages specialized hardware, in the form of "in-network" sequencer (a programmable switch or a smart NIC), to totally order a sequence of messages. In failure free operation, the protocol avoids explicit coordination among the participants, since the ordering is defined by the network itself. NOPaxos ensures that for every message sent to a destination group, either no process receives the message, or all processes receive the message or a notification that the message was dropped. In addition, if two messages are sent to a destination group, every process that receives both messages delivers them in the same order.

To order messages, the component that functions as the network switch keeps a monotonically increasing counter, which it increments each time it forwards a message. As part of the forwarding procedure, the message is automatically tagged with the counter value. These tags allow recipients to deliver requests in the same order and detect out-of-order or dropped requests. If a message drop is detected, recipients need to coordinate to execute a recovery procedure (which recovers the message or assigns a special "drop" message to that sequence number). This broadcast primitive can be used to create a shared log that is constructed by assigning requests to log entries according to the order established by the network.

The advantage of this system is that it allows for a coordination free mechanism to achieve ordering of requests. The disadvantage of this algorithm is that it requires specialized hardware for the sequencer. Moreover, the in-network sequencer is a physically centralized component that may become a bottleneck in the system operation. In the event of a sequencer failure, costly coordination between receivers is required to determine which entries must be marked as "drop". Finally, to ensure a total order, all replicas must receive all the client append operations.

3.2.4 Hydra

As discussed above, NOPaxos [16] uses a single centralized "in-network" sequencer that can become a bottleneck and a source of downtime. Hydra [8] attempts to circumvent these problems by using multiple active sequencers concurrently as opposed to a single sequencer. Hydra uses timestamps taken from loosely synchronized clocks on each sequencer. To guarantee the safety of the algorithm, these clocks must be monotonically increasing.

A client is able to send the operation to any sequencer, the operation is assigned a timestamp taken from the sequencer clock. The order of the operations is based on these timestamps, but because there are multiple sequencers, the system must ensure that after delivering an operation, no future operation will arrive with a lower timestamp. Therefore, the system waits to deliver an operation until all other sequencers have reported timestamps that exceed the timestamp assigned to that operation. In the case where two timestamps are the same, the order is defined by the sequencer's unique identifier.

Waiting for the timestamps of other sequencers to progress introduces a liveness problem. If a message is assigned a timestamp by a sequencer but no other operations arrive from other sequencers, the system will wait indefinitely to get information about their clock values. To solve this, Hydra uses flush messages that are sent periodically by all sequencers and contain the current clock value. With this information, Hydra orders operations that were on hold waiting for higher timestamps from other sequencers. Furthermore, Hydra can optimize the flush messaging process by using client side solicitation, i.e., the subscribers requests flush messages when needed to speed up the delivery of operations. Although this approach bypasses the need to wait for flush messages, it increases the number of messages exchanged for each delivery, and in the worst case, when a subscriber receives an operation, a roundtrip of latency is necessary before it can be delivered.

The advantage of this system is that it no longer relies on a single sequencer to order operations. The disadvantage is that, if the operations are not uniformly distributed across sequencers, clients need to wait for flush messages, which increases latency. Furthermore, Hydra supports only partial ordering for operations.

3.2.5 ZipLog

In all the systems described above, the ordering of append requests is in the critical path of clients, i.e., it is performed after the append operation is invoked and before any read operation can observe the new entry. This introduces a non-negligible latency in the path between producers and subscribers. ZipLog aims at circumventing this limitation. To achieve this, ZipLog orders operations *before* they are issued by clients. This reduces the

latency in the critical path and avoids bottlenecks that may occur when multiple clients use a single sequencer concurrently.

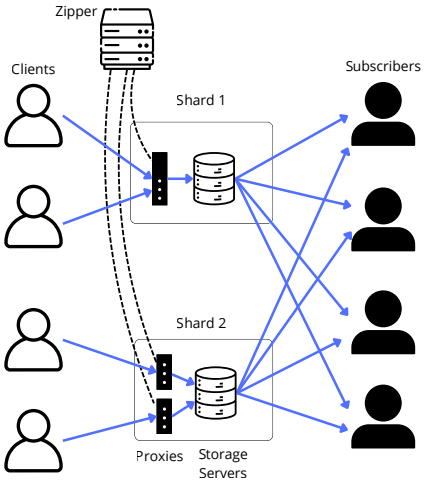


Figure 2: ZipLog System Architecture

The ZipLog architecture, illustrated in Figure 2, includes client processes and storage servers that are organized in replication groups. Additionally, ZipLog assumes that all operations that are sent to a given replica group are mediated by one or more *client proxies*, which help in the process of pre-ordering requests. A replica group of storage servers and its associated client-proxies is denoted as a ZipLog *shard*. ZipLog also uses a component to assist in the ordering of records, referred to as Zipper. However, and unlike previous systems, the Zipper pre-assigns sequence numbers to requests in advance, based on throughput estimates that it periodically adjusts with the help of the client proxies.

Clients are required to register with a proxy before issuing appends to the log. Proxies are responsible for assigning a predetermined sequence number to incoming operations and forwarding ordered operations to storage servers in the same shard. To ensure a consistent ordering of operations, each proxy receives sequence numbers from the Zipper to later assign to incoming records. To achieve this, ZipLog divides time into epochs. Before each epoch, every proxy informs the Zipper how many slots in the log will be necessary to meet the registered client invocation rate. This ensures that each proxy has enough sequence numbers to assign to client operations arriving in the next epoch. The Zipper uses the proxy estimates to pre-assign slots in the log to each proxy, which starts filling these slots with incoming client append operations and forwarding ordered operations to the storage servers in the shard. Storage servers combine operations that arrive from possibly multiple proxies in the shard and use the assigned sequence number to create a single durable, consistent order.

The advantage of this system is that client appends are distributed across multiple proxies instead of a centralized sequencer ordering all the operations. This results in lower latencies, as appends are ordered immediately upon arrival, but still high throughput and scalability because the system does not rely on a single centralized component that operates on the critical path. One disadvantage of ZipLog is that, similarly to all the previously introduced systems, clients that subscribe to the log have to process all the log entries. These systems do not provide mechanisms for subscribers to consume only log entries of interest. This puts more load than necessary on the subscriber, which causes higher latencies to consume the records of interest.

3.3 Supporting Streams in Shared Logs

The shared log designs covered in the previous section have focused on improving system throughput to support a higher rate of appends by using techniques such as batching, sharding, using multiple sequencers and reducing performance penalties during failures. However, none of these systems has been designed to address the challenge of enabling clients to efficiently read new entries from the log. A shared log may be used by multiple data producers, each appending entries related to different contexts. This means that clients may not be

interested in reading the entire log. If clients must parse the entire log to identify the entries they are interested in, they experience unnecessary delays. This limitation can be mitigated if clients are able to skip the entries of no interest to them, i.e., if the shared log provides support for *streams*. This section describes approaches that implement streams in current shared log implementations.

3.3.1 Tango

Tango [3] is a shared log implementation that addresses the problem of ensuring fast playback speed for clients that are interested only in a subset of log entries. To achieve this goal, Tango introduces a new abstraction over the shared log, called a *stream*, which represents a subset of log entries regarding a particular context. Clients interested in reading entries that belong to a given stream can invoke a `readnext` primitive, which returns the next log entry that belongs to that stream while skipping entries from other streams. Clients can read from multiple streams, and because all entries are inserted in a global shared log, all clients always observe a single total order of entries, regardless of the streams they read from.

To support these streams, Tango uses a centralized sequencer that stores the latest entry for each stream. Clients appending to the log use this information to add metadata to each entry, in the form of *backpointers* to previous entries in the log belonging to the same stream. Similarly to Corfu, Tango also has the problem that a client may reserve an entry in the log for a given stream but fails before filling the entry. The use of multiple backpointers allows clients to navigate in the stream even if some entries are replaced by a "no-op", which itself contains no backpointers. When a client wants to read from a single stream, it contacts the sequencer to find the latest entry and then starts reading the log backward, using backpointers, until it reaches an entry it has already read previously. This allows the client to skip log entries regarding other streams.

One advantage of Tango is that it offers a way for clients to read only the log entries of interest without having to playback the entire log. A disadvantage of Tango is that, like Corfu, it relies on a centralized sequencer that can become a bottleneck in the system. Also, backpointers have some limitations, for example, if a client only wants to read a specific entry from a stream, it might need to playback the entire stream to find it.

3.3.2 vCorfu

Like Tango, vCorfu [2] aims to solve the playback speed bottleneck of shared logs. vCorfu also offers the stream abstraction, but instead of using backpointers, vCorfu stores all log entries in different sets of replicas: stream replicas and global log replicas. Similarly to Tango, vCorfu relies on a centralized sequencer that keeps track of not only the next position in the global log replica, but also the next position in each stream replica. When a client wants to append to the log, it contacts the sequencer. The sequencer returns both the next global log position and the next position within the stream of that entry. The client uses this information to write the operation to both the log replicas and then also to the corresponding stream replica(s). Because all entries are still ordered by the sequencer, which issues sequential tokens with the next global log position, total order across streams is ensured.

Clients who are only interested in a given subset of log entries can read from the corresponding stream replica that stores only those entries. Therefore, unlike Tango, vCorfu allows clients to read a specific entry in a stream directly, without having to follow backpointers.

The limitation of vCorfu is that it relies on a centralized sequencer, which limits the scalability of the system.

3.3.3 Boki

Boki [4] is a shared log that, just like Scalog [12], orders client operations only after storing them, but offers a mechanism for clients to read only a target sub-set of log entries. To meet this requirement, log entries are tagged with labels. Entries with the same label operate as vCorfu [2] streams. To implement streams, Boki maintains an index on the client side that maps which entries belong to each stream.

Append operations operate in the same way as in Scalog, with the additional step that after log entries are ordered, the ordering layer provides subscribers with metadata regarding the mapping between sequence numbers and label. Subscribers use this information to update the local index. Subscribers can take advantage of the sequence numbers stored in the index to read only log entries of a given stream.

An advantage of Boki is that, just like Scalog, it enables high throughput for append operations, provides total order across all shards, and, additionally, allows clients to consume only log entries of interest. The main weakness of Boki is the need to perform index updates, where all subscribers are provided with metadata regarding each and every stream. This results in unnecessary information processing by subscribers, as most clients are not interested in all streams.

3.3.4 Eris

Eris [7] is a system that aims to improve NOPaxos [16] by avoiding the need to send all messages to all recipients. Recall that in NOPaxos, all recipients need to receive all messages, both to detect message drops and to deliver messages in order. To solve this limitation, Eris employs a technique called multi-sequencing, where the sequencer keeps a separate counter for each possible destination group. For each ordered message, the sequencer increments the counter corresponding to the destination group and forwards the message with a multi-stamp, which is a set of <group-id,sequence-num> pairs, one for each possible destination group. This allows correct drop detection without requiring receivers to process all messages. Messages can be sent to multiple destination-groups and, in this case, the different counters are incremented atomically, which ensures that messages that are sent to the same set of groups are delivered in the same order. Hydra [8] also uses this strategy to avoid having to broadcast all messages. However, this mechanism only offers partial ordering, as messages sent to different groups may be received in different orders by different recipients.

The disadvantages of this system include the need for specialized hardware and the fact that, as in NOPaxos [16], no single global total order is imposed on all messages.

3.4 Discussion

From the shared log algorithms surveyed above, it is possible to conclude that systems with a centralized sequencer (e.g., Corfu [9], NOPaxos [16]) exhibit a common scalability limitation (i.e., the sequencer becomes a bottleneck), while also suffering from availability outages when the sequencer fails. Scalog [12] attempts to mitigate these problems, by employing a store first order later approach. This solves the availability outages caused by sequencer failures as the storage servers can continue to accept new operations. However, for the scalability limitation, Scalog employs a batching solution, which while increasing throughput compared to previous systems, causes increased latency for append operations. Another alternative is Hydra [8], which aims to solve the mentioned limitations by using multiple active sequencers simultaneously instead of a centralized sequencer. This successfully eliminates the common bottleneck of previous systems. Nevertheless, because each operation can now be ordered by any sequencer, subscribers must wait for information from all sequencers before delivering an operation, which increases latency. Hydra addresses this problem by having the sequencers periodically send flush messages, but if the distribution of operations is not uniform across sequencers, clients have to either be constantly waiting for the next flush message or use client side flush message solicitation. Both of which increase delivery latency. Additionally, Hydra only offers partial ordering, compared to the total order offered by Corfu [9], Scalog [12], and ZipLog.

The only algorithm described that overcomes all the limitations listed above is ZipLog. This system does not suffer from the centralized sequencer problems as clients are distributed to multiple proxies. The system also does not share the latency limitations of Scalog, because operations are immediately ordered upon arrival at the server, without employing any batching. ZipLog also does not have the latency limitation present in Hydra, because clients do not need to hear from other sequencers before delivering an ordered operation, i.e., the sequence number assigned to each operation on arrival represents the final position of the operation in the global order, not being affected by other future sequenced operations. In addition, ZipLog provides global order. A summary comparison of these shared log systems is presented in Table 2.

From the mechanisms surveyed that enable stream support, it is possible to conclude that most depend on a centralized sequencer (e.g., Tango [3], vCorfu [2], and NOPaxos [16] with the Eris [7] approach to enable streams). This places a limitation on the scalability of the system as the sequencer becomes a bottleneck in the system and represents a single point of failure for system operation, as explained before. Another alternative to these limitations is Hydra, using the Eris approach to implement streams. However, this solution introduces the latency problems mentioned previously and also requires specialized hardware. An alternative approach is

Sequencer	Ordering Layer	Order
Corfu	Central	Global
Scalog	Central (replicated via Paxos)	Global
NOPaxos	Central (In network)	Partial
Hydra	Multi	Partial
ZipLog	Multi (off path)	Global

Table 2: Comparison of ordering mechanisms and their limitations

Boki [4], which, like Scalog, employs a replicated ordering layer that allows storage units to continue accepting requests even in the event of an ordering layer failure. Boki allows subscribers to consume only a subset of log entries by leveraging a client side log index. This protocol, like Scalog, suffers from the problem that the latency for each append to become visible to clients is affected by the batching approach employed by the protocol. Furthermore, subscribers receive index updates related to each and every stream, resulting in unnecessary information processing. The following Table 3 summarizes this comparison.

System	Stream Support	Requirements	Limitations
vCorfu	Materialized Streams	Central Sequencer	Appends need to write to multiple replicas
Tango	Backpointers	Central Sequencer	Sequencer recovery requires log scan
Eris	Multiple Destination Groups	Specialized HW	Only partial ordering
Boki	Log index	None	Subscribers need to receive information regarding all streams
Proposed	**Stabilizer**	**None**	**Stabilizer may be a bottleneck**

Table 3: Comparison of stream implementation mechanisms and their limitations

As explained before, ZipLog successfully overcomes the limitations discussed above. However, the inability to implement multiple streams limits the performance of subscribers interested only in a subset of log entries. The next section explains how this problem will be addressed.

4 Streamed ZipLog

This section outlines the design of a shared log that aims at offering low append latency and efficient support for multiple streams. The proposed system extends ZipLog, given that this shared log is, to the best of our knowledge, the system that provides the lowest append latency. Currently, ZipLog has no support for streams, the proposed architecture augments it with additional components to implement this functionality.

4.1 Architecture

The proposed design, illustrated in Figure 3, preserves all components of the original ZipLog implementation. Storage servers are grouped into replication groups (input shards), each associated with a set of client proxies. Clients use the proxies to append entries to the log and can use the subscription interface to receive notifications regarding newly appended entries. The API of the system will be extended to allow entries to be labeled with tags that identify their associated stream(s). Readers can leverage labels to subscribe to specific streams and be notified when new entries with the desired labels are appended to the log.

Streams will not be statically bound to particular input shards. A client is assigned to a proxy of a given shard and can append entries to different streams. Consequently, subscribers need to read from all shards to obtain the complete set of entries belonging to a given stream. The main difference with regard to the original ZipLog architecture is the introduction of a new component class, the *stabilizer*, which assists subscribers in accelerating the delivery of operations. The following section provides a brief description of each component in

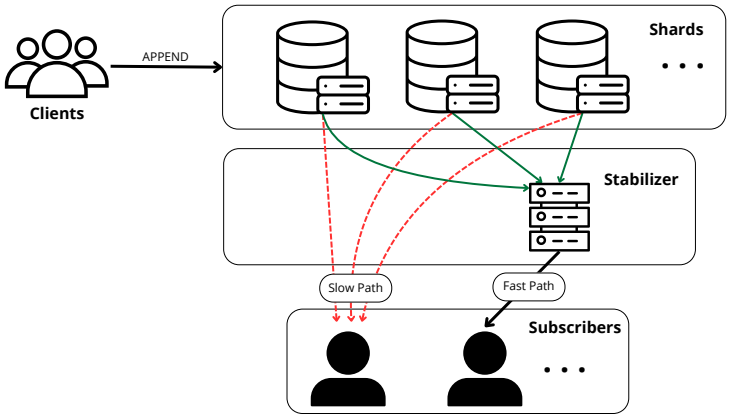


Figure 3: System Architecture

this augmented version of ZipLog, referred to as "Streamed ZipLog". For better understanding, Table 4 outlines the different types of messages exchanged during system operation and their respective contents.

Name	Contents	Sender	Recipient
Record Append Notification	Log entry and associated label	Input shard	Subscriber
Progress Report	Sequence number and label	Input shard	Stabilizer
Flush Message	Sequence number	Input shard	Stabilizer or Subscriber
Delivery Notification	Sequence number	Stabilizer	Subscriber

Table 4: Contents of exchanged messages during system operation

4.1.1 Input Shards

The shards will be responsible for receiving and storing operations arriving from client proxies, receiving subscribe requests from subscribers that specify labels of interest, and transmitting to those subscribers any newly appended records associated with those labels. This message is referred to as a *record append notification*. Furthermore, each shard also reports progress to a stabilizer, i.e., for every received record belonging to a stream, the shard informs the stabilizer with the label and sequence number assigned to the new entry. This notification is referred to as a *progress report*. In effect, shards interact with the stabilizer as though it were a special subscriber, one that is interested only in the metadata (sequence number and label) of newly appended entries.

4.1.2 Stabilizer

The stabilizer receives *progress reports* from the input shards. A stabilizer node will keep track of the last sequence number reported by each shard and, for each stream, a collection of sequence numbers it has not informed subscribers to deliver. With this information, the stabilizer speeds up the delivery of operations by subscribers in the following manner: when a subscriber of a given stream s receives a record with sequence number n from a shard, before delivering the operation, it must ensure that it will not receive in the future a record with a sequence number lower than n from another input shard. Without the stabilizer, this requires the subscriber to wait for all other shards to send either a record for that stream or a *flush message* with a sequence number larger than n (this procedure is referred to as the *slow path*). In the proposed architecture, because the stabilizer receives progress reports regarding all streams, it can determine, more rapidly than the subscriber, that the remaining input shards are already assigning higher sequence numbers. Therefore, the stabilizer can issue a *delivery notification* instructing the subscriber to deliver the operation (this procedure is referred to as *fast path*). Both procedures are illustrated in Figure 3.

It may appear to be a contradiction that in order to prevent a subscriber from receiving *entries* from all streams, this approach relies on a stabilizer that receives *metadata* (in the form of progress reports) for all streams. The key intuition here is that receiving entries is significantly more resource consuming than receiving metadata. As a result, the stabilizer is expected to be able to keep up with subscribers, despite receiving metadata for all streams, since subscribers receive information from only a subset of streams but must also process the corresponding entries. Furthermore, as the following Section 4.2 demonstrates, several refinements can be applied to mitigate stabilizer load and improve throughput scalability.

A key point in our design is that the stabilizer is not required for safety nor for liveness. Even if a stabilizer is overloaded or crashed, subscribers keep delivering entries using the *slow path*. The stabilizer serves only as an optimization by accelerating the delivery of entries using the *fast path*. Without the stabilizer, the system falls back to the *slow path*, which is equivalent to the approach used in Hydra [8], where subscribers need to receive information from all shards to deliver operations.

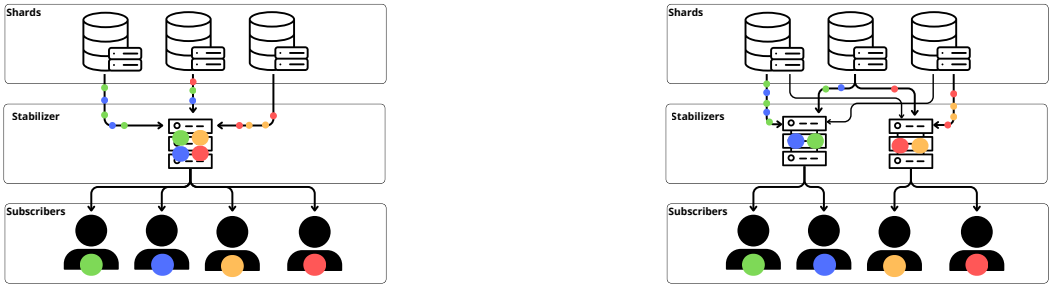
4.1.3 Subscriber

The subscriber informs all shards and the stabilizer of the stream(s) it is interested in. Subsequently, it starts receiving *record append notifications* from the shards and *delivery notifications* from the stabilizer indicating entries that are ready for delivery. An operation is ready for delivery when one of two conditions is satisfied: (1) the stabilizer indicates that the operation can be delivered because it has confirmed that other shards are already receiving operations with higher sequence numbers (fast path); or (2) the subscriber has received, from all other shards, either operations or flush messages with higher sequence numbers (slow path). Note that it is possible that the subscriber receives the *delivery notification* from a stabilizer before it has received the associated record. In this case, the client stores the sequence numbers that are ready for delivery, and upon arrival of the mentioned record, it is immediately delivered.

4.2 Refinements

As mentioned above, having the stabilizer receive progress reports for all the streams in the log can cause a bottleneck that limits the throughput scalability of the system. To address this issue, multiple mitigation strategies are considered, each of which is described in the following sections. These strategies will be implemented and evaluated.

4.2.1 Partitioned Stabilizer



(a) All streams are managed by one stabilizer

(b) Each stabilizer manages a subset of streams

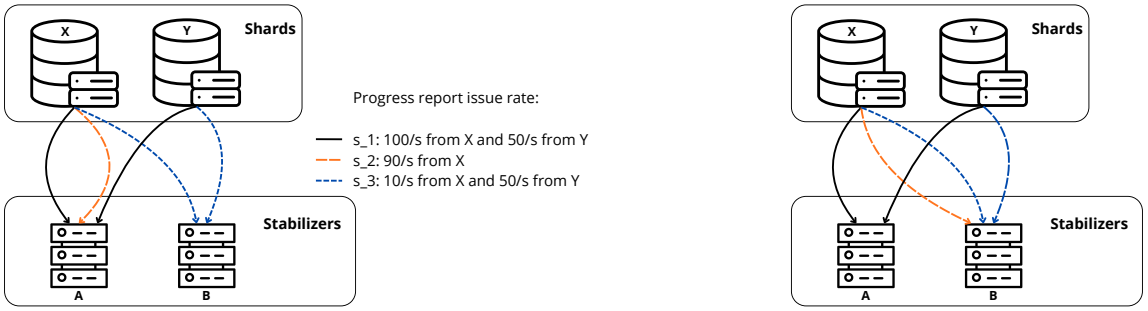
Figure 4: Partitioned stabilizer example

One possible strategy is to partition the stabilizer component, with each stabilizer node responsible for a subset of the existing streams. Figure 4 illustrates an example, where four streams exist, each subscriber is interested in one stream, in 4a all streams are managed by a single stabilizer, whereas in 4b each stabilizer is responsible for a subset of existing streams.

This design provides two benefits: it reduces the load handled by each stabilizer node and enhances fault tolerance of the stabilizer component, since subscribers of streams managed by the remaining stabilizers can continue operating on the fast path even if one stabilizer fails. This strategy enables the system to achieve higher throughput scalability by eliminating potential bottlenecks. However, partitioning the stabilizer causes a slight increase in delivery latency as each stabilizer now receives *progress reports* from fewer streams. Consequently, it takes longer for a stabilizer to gather information from all shards and issue a *delivery notification*. To prevent a stabilizer node from waiting indefinitely for information from shards that are receiving mostly records regarding streams managed by other stabilizers, each shard is configured with a timeout to issue a *flush message* to the stabilizer if it has not sent a *progress report* within that interval. This allows the stabilizer to issue *delivery notifications* even if it never receives *progress reports* from some shards.

This partitioning strategy is expected to be more advantageous in high load scenarios, where a single stabilizer processing all progress reports would result in a bottleneck.

4.2.2 Rate Aware Mapping



(a) Initial mapping of streams to stabilizers

(b) After re-assigning s_2 to stabilizer B

Figure 5: Rate aware mapping example

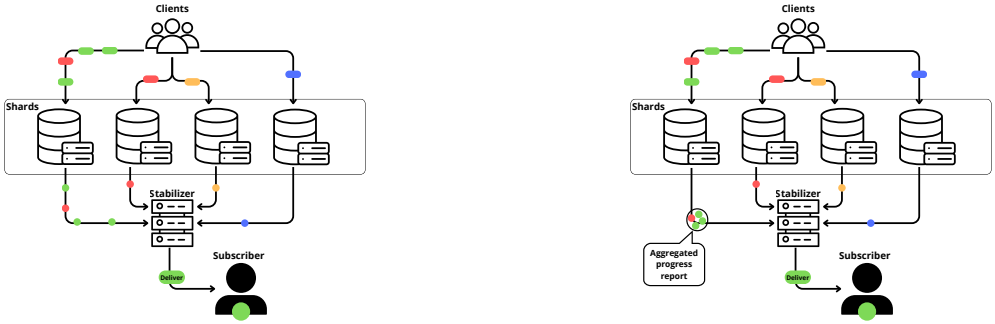
An additional strategy that can be used when using a partitioned stabilizer is to assign streams to each stabilizer using a progress report rate aware mechanism that minimizes the average latency to issue delivery notifications. Consider two shards, X and Y, and three streams s_1, s_2, and s_3. Shard X issues progress reports for all three streams, while shard Y issues reports for s_1 and s_3. Two stabilizers are active: stabilizer A, initially responsible for s_1 and s_2, and stabilizer B, responsible for s_3. Stream report rates are as follows: s_1 causes 100 progress reports/s from X and 50 reports/s from Y; s_2 causes 90 reports/s from X; and s_3 causes 10 reports/s from X and 50 reports/s from Y. This example is illustrated in Figure 5a.

In the initial assignment, stabilizer A is limited by shard Y at 50 reports/s, while stabilizer B is limited by shard X at 10 reports/s. Reassigning stream s_2 to stabilizer B, as illustrated in 5b has no impact on stabilizer A latency to issue the delivery notification, as it is still limited by shard Y. However, it changes the limit of stabilizer B to 50 reports/s by shifting its bottleneck to shard Y. This example demonstrates that the average latency to issue delivery notifications can be improved by mapping streams to stabilizers in a way that leverages information about the rate at which each stabilizer receives progress reports from each shard. In general, this mechanism should employ a mapping that minimizes the difference between the arrival rates of progress reports from shards that issue these at higher rates and from shards that limit the *delivery notification* issuance rate. This strategy is expected to provide greater benefits when this imbalance between these rates is greater.

4.2.3 Dynamic Activation

Another strategy is to activate or de-activate the stabilizer under certain workload conditions. Specifically, when streams are receiving entries at a fast rate and are evenly distributed through the shards, the slow path also can achieve low latency (because new entries, with higher sequence numbers, are received from all shards shortly after entries pending for delivery). This strategy can avoid wasting the resources required for the operation of the stabilizer in situations where it could become a bottleneck and offer minimal improvement in latency.

4.2.4 Batching



(a) Shards continuously send progress reports to stabilizer

(b) Shards can produce aggregated progress reports

Figure 6: Progress report batching example

A further strategy to improve throughput under high-load scenarios, where some input shards continuously send *progress reports* to the stabilizer, is to batch these prior to transmission. Figure 6 illustrates this refinement. The resulting aggregated *progress report* can contain information regarding multiple streams. This reduces the number of messages that the input shards must send and the stabilizer must process, thereby alleviating potential bottlenecks. However, this mechanism should only be employed by some input shards to ensure that the delivery notifications sent by the stabilizer are not delayed as a result. When applied only to the subset of shards that emit *progress reports* at the highest rates, batching reduces message exchanges without impacting the delivery latency of subscribers. This strategy is expected to be more effective in scenarios where progress append rates are unevenly distributed across shards, enabling shards that continuously issue *progress reports* to batch them without introducing latency for subscribers.

4.2.5 Flush Messages

The proposed approach falls back to the slow path upon a crash of the stabilizer. Notably, in the possibility that shards take too long to detect a stabilizer failure and begin sending flush messages to subscribers, delivery latency can increase. Another strategy is to have input shards send flush messages to subscribers continuously rather than only during stabilizer failures. Under normal operation, these messages are redundant with regard to the information provided earlier by the stabilizer, and are thereby discarded by the subscribers. With this approach, the system becomes less susceptible to short-term performance penalties during failure scenarios, but incurs a slightly higher overhead under normal conditions. Depending on application requirements, this can be a beneficial trade-off.

4.3 Expected Gains and Tolerance to Skew

The proposed approach improves upon previous mechanisms to implement streams that relied on centralized sequencers (e.g. vCorfu [2], Tango [3], Eris [7]), by extending the ZipLog protocol, which circumvents the problems related to bottlenecks or single point of failure mentioned before. This approach also overcomes the Boki [4] limitation, where all clients must receive information regarding every stream, allowing each client to only receive data regarding the streams of interest. There is also no latency overhead caused by batching, as ZipLog orders operations immediately upon arrival at the server.

The described approach takes inspiration from the Hydra [8] protocol. This is, to the best of our knowledge, the only system that enables clients to process only a subset of all log entries and is not dependent on a single centralized sequencer, which would otherwise limit the scalability of the system.

Importantly, the proposed design does not share Hydra's latency limitations. In particular, when streams are skewed to particular shards, Hydra relies on either flush message timeouts or client side flush solicitation, both of which are expected to incur higher latency than the proposed approach. In general, the proposed approach with

the stabilizer is not affected by stream skew across shards, since the stabilizer receives information regarding multiple streams from each shard.

For workloads that demonstrate evenly distributed streams across shards and each stream is receiving entries at a high rate, the stabilizer can be de-activated, as explained in 4.2.3. Even with the stabilizer de-activated, the proposed approach still provides global order across shards, rather than partial ordering (as in Hydra), since ordering is established with ZipLog.

Intuitively, streams will most likely not be distributed evenly across the shards due to skewed usage patterns and inherent variability of workloads. The remainder of this section provides a more detailed comparison with the Hydra algorithm under these scenarios. Base Hydra implementation is referred to as *Periodic Flush*, the Hydra implementation with proactive client side solicitation is referred to as *Proactive Subscriber* and the proposed approach is referred to as *Stabilizer*.

In the *Periodic Flush* approach, clients can be continuously overwhelmed by flush messages. The *Stabilizer* approach overcomes this limitation, since the stabilizer issues a *delivery notification* for subscribers only once for each newly appended record regarding the subscribed stream. Although the *Proactive Subscriber* approach also circumvents this limitation, it increases the number of messages exchanged for each delivery, and in the worst case, when a subscriber receives an operation, a roundtrip of latency is necessary before it can be delivered. The proposed *Stabilizer* strategy reduces both the delivery latency and the number of messages exchanged. Table 5 compares the expected latency for a subscriber to deliver an operation after its arrival, across the three approaches, when each stream is skewed towards some shards.

Strategy	**Expected Latency**
Periodic Flush	$T_{\text{timeout}}/2$
Proactive Subscriber	2 Communication Steps
Stabilizer	1 Communication Step

Table 5: Comparison of different approaches in terms of delivery latency after arrival at the subscriber

In the *Periodic Flush* approach, the subscriber must wait for the next flush message from the shards that have not received updates for the subscribed stream. In the *Proactive Subscriber* strategy, the subscriber must request the flush message and wait for the reply. This yields a total of 2 communication steps. In the proposed *Stabilizer* approach, even fewer communication steps are required, as each shard sends the operation to the subscriber while simultaneously issuing a *progress report* to the stabilizer. The stabilizer combines this information together with reports from the remaining shards and issues a *delivery notification* to the subscriber. This results in a single communication step of latency overhead after the arrival of the record at the subscriber.

Another aspect that affects system performance is the number of messages exchanged that is necessary to provide stream support. Table 6 compares how this overhead scales, using as a baseline the fact that, regardless of the approach, shards must always send *record append notifications* regarding a stream to the corresponding subscribers. The table only refers to the additional overhead on top of this baseline, when each stream is skewed towards a single shard.

Strategy	**Messages**
Periodic Flush	$(\#Shards - 1) \times \#Subscribers$
Proactive Subscriber	$2 \times (\#Shards - 1) \times \#Subscribers$
Stabilizer	$\#Shards + \#Subscribers$

Table 6: Comparison of different approaches in terms of number of messages to provide stream support

In the *Periodic Flush* approach, every shard, except the one that sent the *record append notification* to the subscribers, has to send a *flush message* to every subscriber of the stream. In the *Proactive Subscriber* strategy, each subscriber proactively requests a flush message from every shard, resulting in twice the number of messages compared to the *Periodic Flush* approach. In the proposed *Stabilizer* approach, each shard issues a *progress report* to the stabilizer, which then sends each subscriber a *delivery notification*. The proposed solution achieves the lowest number of messages exchanged in order to provide support for streams.

Note that all of the above strategies can optimize the number of messages exchanged necessary to deliver operations. Both the *Periodic Flush* and the *Proactive Subscriber* strategies can use the same flush message to deliver multiple pending operations. Similarly, the proposed *Stabilizer* approach can employ batching, as described in 4.2.4, to further reduce message exchanges.

Importantly, if the system uses a partitioned stabilizer (4.2.1) and the operation belongs to multiple streams, the shard might need to send the *progress report* to more than one stabilizer. Additionally, if the proposed approach uses redundant flush messages (4.2.5), this would increase the number of messages the shard must send.

Critically, the stabilizer is still limited by the shard that issues *progress reports* at the slowest rate, as it requires this information before issuing a *delivery notification*. The refinements presented in 4.2.2 and 4.2.4 leverage this imbalance between shards. The first increases the rate at which the stabilizer can issue *delivery notifications*, while the second leverages the imbalance to reduce both the number of *progress reports* that the shards must send and those that the stabilizer must process.

Finally, the proposed design augments the original ZipLog system as it is expected to allow subscribers to efficiently consume only a subset of all log entries.

5 Evaluation

With the proposed approach implemented, an evaluation will be conducted to determine whether the system efficiently allows clients to consume only the log entries of interest.

5.1 Methodology

The evaluation will be conducted using the proposed system implementation. Experiments will be conducted on physical machines deployed within a single data center. The evaluation will use synthetic workloads to systematically vary and assess different levels of stream skew across shards.

The evaluation will consider four different implementations: (1) base ZipLog implementation; (2) the proposed approach; (3) ZipLog extended with the *Periodic Flush* strategy; and (4) ZipLog extended with the *Proactive Subscriber* strategy. Evaluating all four variants across multiple deployment configurations and multiple workloads would result in an infeasible number of experimental combinations. Thus, only a subset will be selected for evaluation. This subset will be determined based on preliminary results achieved.

5.2 Evaluation Objectives

The evaluation aims at answering the following questions:

- How do streams benefit the base ZipLog implementation? (5.2.1)
- How do the proposed refinements impact the overall architecture? (5.2.2)
- How does the timeout configuration affect the behavior of the system? (5.2.3)
- How does the stabilizer strategy compare to alternative approaches? (5.2.4)

5.2.1 Latency benefits of Streamed ZipLog

To motivate the need for streams in a shared log, the evaluation will begin by comparing the proposed solution to the base ZipLog implementation. By allowing subscribers to only read log entries of interest, a decrease in record's end-to-end latency is expected (i.e., from the time the record is appended to the log until the subscriber observes and processes it). For a given stream, the improvements are expected to be inversely proportional to its popularity, as this allows the subscriber to skip a greater majority of records.

Refinement	Target Workload
Partitioned Stabilizer	High throughput so that a single stabilizer would result in a bottleneck
Rate Aware Mapping	Imbalance between append rate to streams managed by each stabilizer
Dynamic Activation	Well distributed streams with high append rates across shards
Progress Report Batching	Imbalance between append rate at the shards

Table 7: Refinements and target workloads

5.2.2 Evaluation of Architectural Refinements

The second point of the evaluation will be to determine the best parameterization for the architecture refinements mentioned in Section 4.2. The goal is to determine the workload characterization thresholds that should trigger each of them in order to obtain the highest performance. Specifically, each refinement will be assessed under the described target workload summarized in Table 7 to quantify its impact on performance. The evaluation will assess the advantages of each in isolation and in combination to identify potential interactions. The goal is to define the conditions under which a given refinement improves performance and those under which it can introduce overhead.

5.2.3 Effect of Timeout Configuration

Another aspect of the evaluation will assess how the timeout values that trigger flush messages affect system performance in two contexts: (1) flushes sent from shards to subscribers when the stabilizer fails, and (2) flushes sent from shards to stabilizers when a partitioned stabilizer is used (4.2.1).

Flush messages sent from shards to subscribers prevent subscribers from waiting indefinitely in the presence of stream skew across shards, as explained in 3.2.4. Similarly, flush messages sent from shards to the stabilizer reduce delivery latency when the stabilizer is waiting for progress reports from shards that mostly receive appends regarding streams managed by other stabilizers.

Importantly, short timeouts can negatively impact performance, as shards may send flush messages prematurely instead of waiting for subsequent appends.

5.2.4 Comparison with Alternative Strategies

Finally, the evaluation will assess how the proposed system compares with existing strategies that implement streams. Two of the previously mentioned state-of-the-art techniques (*Periodic Flush* and *Proactive Subscriber*) will be implemented on top of the base ZipLog algorithm. Subsequently, both will be evaluated against the proposed approach. The proposed *Stabilizer* strategy is expected to yield improvements proportional to the stream skewness across shards. The more skewed a stream is, the worse the performance will be without the stabilizer, as the subscriber has to wait for information from the shards receiving fewest operations regarding that stream. By receiving a constant flow of information from all shards, the stabilizer can speed up the delivery of operations by directly informing subscribers, rather than requiring them to wait for information directly from the shards regarding the specific stream of interest.

6 Work Schedule

The Gantt chart in Figure 7 illustrates the planned schedule for various tasks related to the MSc dissertation.

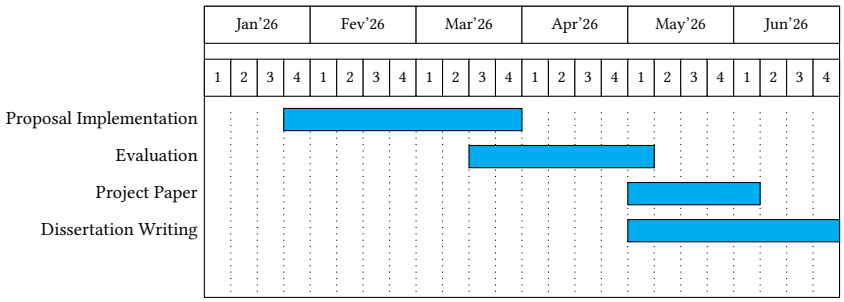


Figure 7: Planned Schedule

7 Conclusions

Shared logs are a crucial component in distributed systems. By maintaining a persistent and totally ordered sequence of records, they enable coordination among multiple servers. However, when clients must process all log entries instead of only those relevant to their interests, scalability is hindered. Unfortunately, existing shared log implementations that offer support for streams rely on mechanisms that include either centralized sequencers or complex coordination that limit the scalability of the system. This report surveyed the state-of-the-art shared log and replication systems. Their implementations were described, detailing the advantages and limitations of each approach. A novel approach to efficiently offer support for streams in a shared log was introduced. Finally, the evaluation methods planned to assess the quality of the resulting system were presented and the scheduling for future work was outlined.

Acknowledgments. I am grateful to Rafael Soares for the discussions and feedback regarding my work. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025 and GLOG (financed by the OE with ref. LISBOA2030-FEDER-00771200).

Bibliography

[1] F. Schneider, "Replication management using the state-machine approach," in *Distributed Systems, 2nd Edition*, ser. ACM-Press, S. Mullender, Ed. Addison-Wesley, 1993, ch. 7.

[2] 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 *NSDI'17*, Boston, MA, USA, Mar. 2017.

[3] 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 *SOSP '13*, Farminton, PA, USA, Nov. 2013.

[4] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *SOSP '21*, Virtual Event, Germany, Oct. 2021.

[5] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI'06*, Seattle, WA, USA, Nov. 2006.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *ATC'10*, Boston, MA, USA, Jun. 2010.

[7] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *SOSP '17*, Shanghai, China, Oct. 2017.

[8] I. Choi, E. Michael, Y. Li, D. R. K. Ports, and J. Li, "Hydra: Serialization-Free network ordering for strongly consistent distributed applications," in *NSDI'23*, Boston, MA, USA, Apr. 2023.

[9] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, Dec. 2013.

[10] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, Jul. 1990.

[11] R. V. Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04*, San Francisco, CA, USA, Dec. 2004.

[12] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. V. Renesse, "Scalog: Seamless reconfiguration and total order in a scalable shared log," in *NSDI'20*, Santa Clara, CA, USA, Feb. 2020.

[13] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *ICSE'76*, San Francisco, CA, USA, Oct. 1976.

[14] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The primary-backup approach.* ACM Press/Addison-Wesley Publishing Co., 1993, p. 199–216.

[15] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[16] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to paxos overhead: replacing consensus with network ordering," in *OSDI'16*, Savannah, GA, USA, Nov. 2016.