Shared Logs With Support for Multi-Consistency

(Extended Abstract of the MSc Dissertation)

FRANCISCO CARVALHO, Instituto Superior Técnico, Portugal
SUPERVISOR: LUÍS RODRIGUES, Instituto Superior Técnico, Portugal

Shared logs are a key component of modern distributed systems. Typically, logs record a totally ordered sequence of events and/or commands that need to be (or have been) executed in a distributed system. Processing events in the total order defined by the log simplifies the task of providing strong consistency guarantees to the application, such as linearizability or serializability. Unfortunately, processing record entries in serial order is also a source of bottlenecks and high latency, in particular when the system experiences delays in filling some records. One way to circumvent these limitations is by exploring application semantics, for instance by allowing operations that are known to be commutative to be processed in parallel and by allowing some entries to be processed even if some of the previous entries have not been filled. In this work we show how hybrid consistency models can be added to shared logs, namely, we present the design and evaluation of RB-Log, a shared log with support for Red-Blue consistency with Immediate operations. An experimental evaluation shows that RB-Log can reduce the latency of blue operations as long as red operations do not exceed 50% of the workload.

Additional Key Words and Phrases: Distributed Systems, Shared Logs, Consistency, Multi-Consistency

1 Introduction

Shared logs are a key component of modern distributed systems. Several production-grade applications have been built using shared logs, such as Apache Kafka[1] and Facebook's LogDevice[2]. The simplicity of the log abstraction - which defines a totally ordered sequence of operations - makes it an appealing choice for building distributed applications that require strong consistency.

Unfortunately, enforcing strong consistency has a negative impact on performance, an important aspect in large scale systems. This forces developers to make a choice. In some applications, strong consistency is a strict requirement to ensure application correctness. In other applications, the semantics of the operations allow the programmer to use a relaxed consistency model, reducing coordination costs and improving performance.

Applications where different operations have different consistency requirements also exist. If a log only offers a single consistency level, developers must either opt for using strong consistency for all operations, sacrificing performance even when the application semantics does not require it [10–12], or, instead, use a weakly consistent system and then enforce the required consistency requirements at the application level, which can be expensive and error prone [5]. Hybrid consistency models offer a middle ground between both approaches, allowing applications to define which operations are required to use strong consistency, without significantly impacting the performance of the remaining ones.

The work presented in this thesis is based on the observation that none of the existing state-of-the-art multi-consistency systems leverages the benefits of shared logs. We hypothesize that a shared log can be a suitable building block for these applications, as long as the log offers support for hybrid consistency. This approach can provide a simpler architecture, while yielding performance gains, namely by offering lower latency and higher throughput. Thus, we propose and implement techniques to support causally consistent operations in a shared log, maintaining global order, but allowing causal operations to be processed by the application as fast as possible.

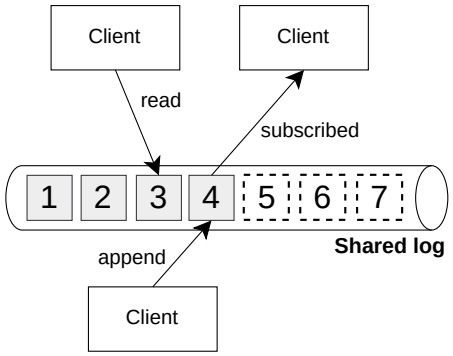

Fig. 1. Shared log.

2 Related Work

We briefly survey some of the most relevant works in the area of shared logs and hybrid-consistency systems.

2.1 Shared Logs

A shared log can be viewed as a single append-only list where multiple clients can concurrently append and read positions. As there can only be one operation per log position, the system guarantees a total order. Total order enables applications to maintain a detailed timeline of events within the system, simplifying tasks such as statistical analysis or reconstructing state after a failure, making them appropriate for building storage solutions. They are also well-suited for message queues, handling and ordering requests flowing from different systems with fault tolerance guarantees.

Shared logs provide two fundamental operations: append, which sends a new operation to the log and assigns it a position; and read, which allows a client to retrieve the operation stored at a given log position.

Among the various existing systems, we highlight Corfu.

Corfu [4] is a shared log system focused on providing strong consistency and performance using clusters of flash-memory for storing entries. Corfu uses a client-centric model in which clusters

Authors' Contact Information: Francisco Carvalho, franciscojcarvalho@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal; Supervisor: Luís Rodrigues, ler@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal.

only store the entries and clients are responsible for appending and replicating the data. Since storage servers are limited to data storage, both logic and networking can be offloaded to FPGAs attached to SSDs, replacing traditional servers.

The log is an abstraction implemented as a set of append-only positions distributed across storage servers. As such, clients have to maintain a mapping of log positions to the physical entries in the flash-memory (and their respective replicas), called a projection. Clients can append data directly to the last available log position, which involves iterating the log from the beginning until an available position is found. This is, however, inefficient as if multiple clients attempt to write at the last position, all of them except for one will abort. To minimize such append contention, Corfu uses a sequencer, a single intermediary that clients contact to reserve a free position in the log. After a position has been reserved, the client is able to append the data to the corresponding shard and its replicas using a replication model based on chain replication [13].

While using a sequencer and reserved positions avoids contention, the sequencer itself becomes a bottleneck and may introduce holes in the log in the presence of network partitions or client failures. This is problematic for clients processing the log under state machine replication, as they cannot advance to the next entry without applying the one that is missing. To address this, Corfu allows clients to either mark entries as junk when they suspect a possible hole in the log, or to help the system in completing an entrie's replication protocol.

In the event of a replica failure or the addition of a new server to the system, the client's projection must be updated to reflect the new topology. To ensure that all clients share the same projection, its versions are tracked using a Paxos-maintained epoch number, which is included in all messages sent by the clients. Thus, changing the projection implies starting a new epoch. Projection changes are launched by a client. In order to prevent race conditions with pending writes, the client first attempts to seal, for the current epoch, the flash units whose ranges of positions are about to change to another server. When a storage server is sealed, it will reject any new messages, rendering the system unavailable during reconfiguration. After all the necessary servers are sealed, the client tries to propose its new version of the projection to the paxos nodes. If a new cluster needs to be added, Corfu performs one reconfiguration to add the new entries to the projection. In case of replica failure, a first reconfiguration is initiated to remap the unfilled entries of the failed replica to a new server, ensuring availability of the system. Then, in the background, the data stored in the backups will be copied to this server. When finished, a new reconfiguration is initiated to transfer the mapping of the failed entries from the backup to the new server.

2.2 Multi-Consistency

As discussed in section 1, application developers traditionally had to choose between strong consistency and high throughput. Strongly consistent systems incur performance penalties due to the coordination required for maintaining a total order, while eventually consistent systems achieve higher throughput by minimizing coordination at the cost of stale data and complex reconciliation. In some cases, however, only a subset of operations truly requires strong consistency to ensure correctness. Multi-consistency systems address this balance by enforcing strong consistency where needed while allowing other operations to benefit from weaker consistency.

Among the existing multi-consistency systems, we highlight LazyReplication [9] and Gemini [10].

Lazy Replication [9] describes a system for replicating state across multiple servers. Clients issue operations against a single replica, which are then propagates it to other replicas. The causal past is, in this system, represented as vector clocks. In a vector clock structure, each entry represents the last observed operation originated from a specific replica.

The system supports three different types of operations: *Causal*, *forced* and *immediate* operations.

Causal operations do not require coordination between replicas to be accepted: any replica can independently accept an operation, which is then propagated, in background, to others using a gossip procedure. Clients send operations to their preferred replica. Each operation includes a vector clock representing the causal past of their respective client. Once received, the operation is placed in a *log* – a queue-like structure to hold pending operations. Additionally, the replica increments its own vector clock, placing the operation in its causal past. The append process is then finished and the client adds the newly-appended operation to its vector clock. Despite the append process being completed, the operation will remain in the *log* until it is ready to be applied in the local state, i.e. when all of its causal dependencies have been locally applied. The applied operation will then be propagated via the gossip protocol to the remaining replicas, where, similarly to client appends, it is applied to local state when all their dependencies have been applied. Causal consistency thus allows the system to provide high-availability for operations that do not require strong semantics.

Forced operations require a total order among all other forced operations, while still preserving causal order with causal operations. This is implemented by extending the vector clock with an additional shared entry that is incremented with each new forced operation, thereby establishing a total order among them. Each replica must apply all operations with a lower value in this component before executing a given forced operation. To avoid concurrent updates having the same number, only one replica at a time can be responsible for incrementing this new entry. In this system, write access to the strong component is regulated using a Primary-Backup model with view changes. The primary proposes a value for the pending forced operation, committing it after a quorum of acknowledgments. The client is then informed of the successful completion and the log record is propagated to the remaining nodes via gossip. Upon view change, a new coordinator is elected and must determine the last issued forced operation. It gathers information from a majority of replicas, computing the highest sequence number assigned between all previously committed operations. This ensures that the new primary can safely assign sequence numbers from this point to new operations.

Immediate operations impose a total order in regard to all operations (*causal*, *forced* and *immediate*). These operations will slow down or even block the *causal* operations as they require a temporary halt of requests due to the total order between all operations.

Immediate operations are also executed by the Primary-Backup view (assuming the view includes all replicas) and involve system-wide synchronization using a 3-phase protocol.

Gemini [10] is a geo-replicated storage system with support for multiple levels of consistency. The aim of this system is to distribute an application's state across geographical locations so as to provide low latency local access to clients. To do so, each local site maintains a (possibly stale) copy of the application's state that is updated each time it receives an update, either from a remote site or from a local client. Ensuring a strong consistency model under these conditions requires replica synchronization, which is expensive. Gemini tries to reduce the impact of strong consistency by using it only when necessary, preferring causal consistency instead. In order to do so, the authors introduced the RedBlue consistency model.

Under RedBlue there are two types of operations, named with colors that reflect two consistency levels:

- Blue operations operate under causal consistency. Blue operations can be immediately applied as long as all their causal dependencies have already been applied.
- Red operations execute under strong consistency, requiring a total order of execution among red operations (they also require their causal past to be present before execution). This, in turn, requires cross-site synchronization and is thus slower than blue causal operations.

The weaker consistency of blue operations allows for different replicas to play the operations in different orders, something that can lead to a permanent state divergence. To prevent this, only globally commutative operations can be blue, meaning the execution order of any two blue operations always results in the same state. Furthermore, an operation may only be made blue if its concurrent execution does not break any of the application's invariants.

The global commutativity requirement, while necessary to ensure state convergence, restricts the set of operations that can be classified as blue. However, in some cases, an operation may not be commutative, yet their computed state transition might be. Taking the example from the paper, in a bank application, a calculation of due interest is not commutative as it depends on the possibly stale balance, but the operation that resulted from the calculation can be written as a deposit which is commutative and can be a blue operation. This concept is implemented by separating each operation into two parts

- The Generator operation computes the effects that the operation will have on the current state, without changing it.
- The Shadow operation applies the computed changes to the local state.

The concepts explained above are implemented in Gemini as follows: All locally appended operations are totally-ordered, while remote operations are applied under the RedBlue consistency model. Each client sends a request to a local proxy server that will execute the generator operation. Once completed, the shadow operation is sent to a local *concurrency coordinator* that will check whether the operation is admissible under RedBlue consistency. This admissibility requirement determines whether the local replica has applied all the required causal dependencies for that operation. If it passes, it is then forwarded to a *data writer* that will apply the operation it into the state.

Similarly to Lazy Replication, the concurrency coordinator relies on timestamps in the form of vector clocks to check if operations are admissible. Each entry of the vector clock represent the latest applied operation originated from remote sites at the local replica. The vector clock holds an additional entry aimed at tracking the total order between red operations. To ensure that no two red operations obtain the same timestamp, only one replica is able to append red operations at a time. This is regulated via a token passing scheme among replicas. A replica holding the token is allowed to append red operations and replicate them. For correctness, only one replica at a time may hold the token at a time, being periodically passed along to other replicas.

3 A Multi-Consistent Log

In this section, we propose a set of mechanisms to implement a shared log with multi-consistency support. Specifically, it supports red operations, which must be totally ordered among other red operations; blue operations, which only need to be causally ordered; and immediate operations, which must be ordered against all other operations. The advantages of blue operations are twofold: first, blue operations may be processed even if the log is incomplete, as long as all its causal dependencies are met. Second, they may also be processed concurrently, improving processing parallelization. Our goal is to reduce the observed latency when consuming weak operations and improve processing throughput.

3.1 System Model

We assume an asynchronous system operating under the crash failure model, in which multiple processes interact with each other via a shared log. Clients can be *producers*, that append operations to the log, or *subscribers*, that read the log and apply operation following the state machine replication model (a process can be both a producer and a subscriber). We assume that the log defines a total order on all operations. We also assume that producers can write concurrently to different log entries. As such, a given log entry may become visible ahead of those ordered before it. The log itself is implemented using a set of geo-distributed and replicated servers. Due to the distributed nature of the log implementation, log entries may become visible to different clients at different points in time (and in different orders). However, eventually all entries become visible to all clients in the same total order defined by the log. Instead of building a novel log from scratch, we aim to design mechanisms that can be applied to existing logs.

3.2 Operation Types

The system supports three types of operations, namely *blue*, *red* (equivalent to the RedBlue consistency) and *immediate* operations (equivalent to the LazyReplication operation), as described below.

- **Blue Operations:** Blue operations execute under the causal consistency model, which allows them to be applied immediately (by subscribers) as long as all operations in their

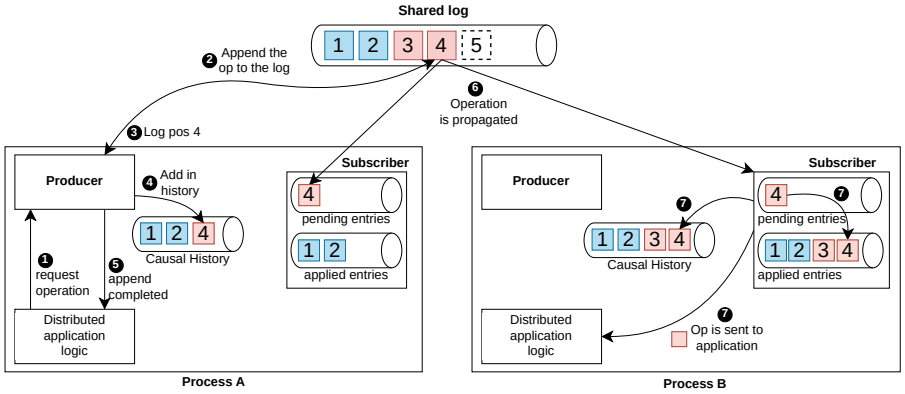

Fig. 2. Operation with two replicas in different locations.

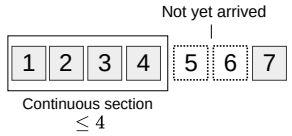

Fig. 3. Example of a local log with a continuous section and two late operations.

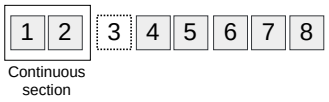

Fig. 4. Example of how a late arrival of a single operation may make the explicit section much bigger.

causal past have been applied. This allows a blue operation to be applied even if some of the previous entries in the log are not yet visible. Because different operations may become visible to different subscribers at different points in time, subscribers may apply blue operations in different orders. Much like in RedBlue consistency, we assume that blue operations are "globally commutative" [10], such that all subscribers converge to the same state. Therefore, causal consistency improves performance by allowing out-of-order execution while global state convergence ensures it can do so safely (by preventing permanent state divergence).

- **Red Operations:** Red operations inherit the causal requirements of blue operations while adding the restriction that they must be applied by all subscribers in the same total order. The total order of these operations are defined by the underlying shared log.
- **Immediate Operations (IM):** are a stricter version of the red operation, in which the operation needs to be ordered against all other operations. As a result, IM operations impose a barrier on the system, restricting the execution of sub sequentially ordered blue and red operations. Such operation can be useful when there needs to be an "atomic" operation that affects the entire system (such as reconfigurations).

3.3 General Architecture

Clients maintain a *causal history* of all operations they have written or read from the log. For clarity of presentation, consider the causal history as a list of log indexes. Note that a client that is a pure producer only keeps the log entries of the operations it has added to the log. A client that is also a subscriber additionally keeps a record of all operation it has read and applied.

When a client executes an append operation to the log, it tags the operation with its causal history. The client then waits for a position to be assigned to the new operation and adds it to its causal history. This concludes the append operation.

When an operation becomes visible (i.e., a subscriber is notified that a given log entry has been filled), the subscriber activates a *watcher* for that entry. The watcher waits for the entry to be *ready* to be applied, applies the operation, and adds its index to a set of *applied entries*, which records all entries that have already been applied by that subscriber.

The condition that makes an entry *ready* depends on the type of operation. If the operation is tagged as red, it becomes ready once all causal dependencies and all red operations ordered before it have been added to the *applied entries* set. A blue operation, in contrast, only requires that its causal past be present in the *applied entries* set. Immediate operations, however, require all preceding operations to be applied and none of the subsequent ones.

Because the log is geo-distributed, entries may not become visible in the order defined by the log. Entries written by local clients may be visible immediately to subscribers in the same region, while entries from remote clients with higher propagation delays may arrive later, even if they are ordered earlier in the log. This explains why an operation that is already visible may need to wait before being applied.

The overlaying distributed application that uses the shared log is agnostic to the log's internal workings and multi-consistency support, only issuing operations to the log and receiving them.

Figure 2 illustrates the operation of the system with two processes that are both subscribers and producers. Blue operations are represented in blue and strongly consistent operations in red. The number in each operation corresponds to its position in the shared log, and the applied-entries list tracks the operations delivered to the application. As described earlier, when Process A requests an operation to its producer (❶), it will send it to the log where the operation is ordered in an empty position (❷). The log then replies with its log position that is promptly appended to the causal history

of the process, concluding the append (❸,❹,❺). The new entry will eventually become visible and be sent to the subscribers. In ❻ the appended operation 4 becomes visible at A, but as it is a strong operation, it must wait for operation 3 to become visible so as to not skip any possible red operations. It thus remains in the *pending entries* list until all its dependencies have been applied. Operation 4 also reaches Process B. In this case, as 3 is already visible and applied, the operation can be immediately applied.

If operation 4 was marked immediate instead, Process A would still be unable to apply it. Nevertheless, the behavior would differ for a hypothetical weak operation 5 arriving at Process A: in that case, it could not be applied until 4 had been applied, despite being a weak operation.

This architecture has several advantages over other multi-consistency systems. It is simpler, and the log abstraction inherently provides strong consistency through its total order. Supporting strong and immediate operations thus only requires sequential log processing, whereas other solutions rely on token passing, consensus, or view changes. Furthermore, because entries are totally ordered, we can leverage this property to reduce the number of causal dependencies, which will be discussed in the next section.

3.4 Enforcing Causality

While causal operations depend only on previously seen operations, they require finer-grained dependency management, which can introduce expensive processing costs and undermine the benefits of weak consistency. Previous approaches have proven inadequate for our use case: encoding all seen operations is prohibitively expensive; using a single sequence number introduce fake dependencies; and vector clocks incur high costs from maintaining client membership views. Therefore, we propose a new solution to reduce fake dependencies while controlling metadata size.

This optimization is based on the observation that although one cannot escape from the "holes" in local logs due to missing operations, one can assume that, eventually, a section of this local log will become fully applied, creating a continuous area. We can leverage this by building a timestamp with two components. The first component tracks dependencies below the continuous section of the local log, using a single scalar. The second component is reserved for operations that were executed in incomplete sections (to which not all operations with lower ordering have arrived). This approach allows the system to cut the number of explicit dependencies while still expressing fine-grained dependencies. It also keeps the client's view of the system consistent, as all operations on the continuous section have already been observed and are thus causal dependencies.

Figure 3 illustrates this technique. Consider the local log at a given replica. Grayed out boxes represent applied operations at the replica, while dotted ones correspond to operations yet to arrive. The numbers in each operation represent the total order assigned by the global shared log. The causal dependencies for a new operation to be appended in this current state would be represented as $\{4, \{7\}\}$, where 4 denotes the continuous section of the log, which includes all elements lower than or equal to 4, and the set denotes the explicit dependencies outside the continuous section. If operation 5 was present, the dependency would have been represented as $\{5, \{7\}\}$.

One drawback of this implementation is the reliance on the ability of the client to fill all the holes in their log. This means that delays in a single operation may increase the number of explicit dependencies, defeating its purpose. Take Figure 4: due to missing operation 3, the continuous section is much smaller. As a result, any new operations sent to the log from this client at that instant would have $\{2, \{4, 5, 6, 7, 8\}\}$ as a timestamp.

To avoid the unbounded growth of the non-contiguous section, we introduce a ceiling to the number of explicit operations in the timestamp. If the configured threshold, determined by a variable λ, is surpassed, clients will advance the uniform part of the timestamp to cover the difference between the threshold and the number of operations. As a consequence, some operations will depend on operations not observed by the client, but, in turn, we guarantee the number of dependencies stays controlled.

The optimal length for this upper bound is workload dependent, and the value can either be static or change dynamically according to collected metrics.

4 RB-Log

This section introduces RB-Log, a shared log prototype with multi-consistency support. Instead of building a shared log from scratch, we implement RB-Log as a layer that sit on top of a pre-existing log implementation. Thus, the RB-Log design decisions can be applied to different implementations. For the prototype, we have implemented RB-Log as an extension of the ZipLog [6] system.

We begin by defining the general architecture of ZipLog in Section 4.1. Section 4.2 describes the RB-Log extension and how it supports Red, Blue and Immediate operations.

4.1 ZipLog

ZipLog [6] is a shared log system that establishes a total order by serializing entries to clients in advance. Time is divided into epochs, during which clients are allocated operations proportional to their requested append rate. The number and distribution of reserved entries is adjusted based on the system's observed workload at the start of each epoch. The version of ZipLog used in this work was implemented in C++ and used ZeroMQ[3] sockets for network communication.

ZipLog is comprised of four components:

- **Ordering server** is a logically centralized component responsible for collecting the producers' rates and forwarding them to the storage servers. It also functions as a naming service, distributing the IP addresses of all components.
- **Storage servers** serialize and store operations received from producers according to the pre-assignments. Storage servers may be sharded and/or replicated. They also iterate their corresponding section of the log and broadcast the newly-appended operations to registered subscribers.
- **Client stubs** (or producers) append new entries to the log at their requested rate. If a producer is unable to maintain the pre-determined rate, it instructs the storage server to mark the corresponding entries as junk (NO-OP). In ZipLog,

the producer is implemented as a dedicated stub that runs in a separate thread and bridges an application with the shared log. The application calls the method `insert` which queues a given operation to be sent to the log. Upon completion, the application is notified via a shared variable updated with the operation's assigned log position.

- **Subscribers** concurrently receive entries from all shards using multiple "poll" threads, which insert them in a log-like shared structure. Dedicated "processing" threads then iterate over the log to apply entries in order.

4.2 RB-Log: An extension of ZipLog

RB-Log is a superset of ZipLog that provides hybrid consistency. To achieve this, several modifications were made to the original codebase. This section first presents an overview of the extended system architecture, followed by a detailed description of how each operation type is supported in practice.

4.2.1 Producers. As in ZipLog, operations are still pre-assigned for each client, which appends them at a fixed rate. However, each operation is also tagged with its type and any causal dependencies, if applicable.

Additionally, we introduce a new type of client, the Producer-Subscriber, which, as the name suggests, combines a ZipLog producer with a subscriber and behaves similarly to the example in Section 3.3. Unlike regular producers, Producer-Subscribers must wait for an appended operation to be processed by their subscriber component before issuing a new operation. If this waiting time exceeds the operation send interval, the stub sends junk (NO-OP) entries. Consequently, the causal past of each new operation includes the subscriber's entire causal history rather than only the operations previously sent.

4.2.2 Subscribers. We now focus our attention on the subscriber component and how processing is done in a multi-consistency scenario. As in the previous implementation, once an entry reaches a subscriber, it is placed by a poll thread into the local log where processing threads will be able to process it. These processing threads continuously iterate the log by sequence number and process the entries when they are deemed ready. In RB-Log, the condition for an entry to be ready for application now depends on its operation type: Blue operations can be applied out-of-order, while Red and Immediate operations require execution in log order. To take advantage of the lower application constraints of blue threads, the subscriber employs two types of processing threads: one dedicated to sequentially processing Red and Immediate operations (continuous thread), and one for the concurrent processing of Blue operations (blue thread). When an entry is ready to be processed, it is sent to the "application logic" by the use of a callback function. Such function then inserts it into a list of processed operations, which is used to determine the causal past for future operations. In case of Producer-Subscriber clients, the producer component will see that the entry has been processed and can now send the next operation.

The remainder of this section explains in more detail how RB-Log supports the application of each operation type.

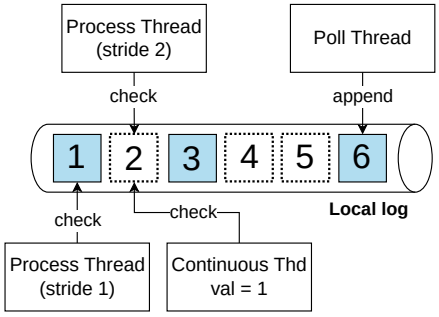

Fig. 5. Architecture of the subscriber's poll and process threads.

4.2.3 Red Operations. Red operations execute under strong consistency and require all previous red operations to be applied ahead. In this implementation, RB-Log uses a stronger version of Red operations, requiring the log to be processed up until the GSN of the current operation. This decision steams from the (current) impossibility of the subscriber to know the type of each operation before their arrival.

Red operations are applied using a single continuous thread. This thread advances to the next entry only after the currently monitored operation has been processed or has arrived as a Red or Immediate operation, thereby tracking the continuous section of the local log and enabling the application of Red operations.

Figure 5 shows a subscriber's local log and the different threads that interact with it. The Continuous thread is shown halted at the unarrived operation 2, which marks the start of the noncontinuous section of the log.

4.2.4 Blue Operations. Blue operations execute under weak consistency. Whenever a blue operation is produced, it is tagged with its causal dependencies. When a client is a pure producer, its causal past is the subset of all operations previously appended by it. In that case, we can express all the causal past using a single number. As for Producer-Subscriber clients, the casual past corresponds to all operations produced or applied. In that case, we must explicitly state the sequence number for each dependency.

Blue operations are processed by a Blue thread once their causal dependencies and all preceding IM operations have been applied. Blue threads operate the same way as the continuous thread, but in case the operation their currently watching isn't available for processing (either because it hasn't arrived or does not have all dependencies) it looks ahead the log for applicable Blue or junk operations. Since Blue operations can be processed concurrently, several Blue threads iterate over a unique partition of the log (which ensures that any given entry may only be applied by one given thread). To ensure processing threads are only iterating a section of the log where there might be entries to process, they only iterate up to the maximum GSN received by all processing threads.

Figure 5 illustrates how blue threads interact with the subscriber's local log. Two Blue threads are shown, each assigned a distinct stride

corresponding to its partition (e.g., the thread with stride 1 only processes odd GSNs, while the other processes even GSNs). These threads may only look ahead up to operation 6, which is the highest registered operation.

4.2.5 Causal Dependency Management. In order to keep the number of finned-grained dependencies low, RB-Log implements the hybrid dependency tracking optimization described in section 3.4.

Consequently, it is necessary to compute the continuous section of the log, which, as previously mentioned, is given by the position of the subscriber's Continuous processing thread. When preparing to send an operation, clients will thus query the continuous thread to determine which operations from the processed list should be included as fine-grained dependencies.

RB-Log also implements a configurable ceiling to the number of explicit operations. As described in section 3.4, whenever the ceiling is reached, the producer trims the section to cover the difference, possibly creating fake dependencies.

4.2.6 Immediate Operations. Immediate operations are totally ordered against all other operations, acting as a distributed barrier. These operations can only be made ready once all previously ordered operations have been applied. Furthermore, they block the processing of all subsequently ordered operations until their execution, regardless of type. This last condition, apart from having the ability to stop the blue operation processing, will require that subscribers know ahead of time if an operation is of type immediate (and thus prevent any new appends after it while it remains unprocessed).

In previous systems, IM operations were implemented using consensus [9]. In contrast, RB-Log leverages the pre-assignment scheme to eliminate this requirement. It supports immediate operations by defining a new stub in each client. This stub behaves as a regular client, requesting a specific rate to the ordering server and marking operations as junk when an operation cannot be produced in time. This design enables the system to distinguish IM from non-IM operations ahead of time, in the same way it differentiates operations from distinct clients.

Additionally, up until now, the subscriber was agnostic to operation pre-assignment, only learning about a particular GSN when the real operation arrived. This behavior had to be changed so that it could identify which slots are immediate. Subscribers now receive and compute the pre-assignments from the ordering server, marking the IM operations in their log. Processing threads were also modified to ensure that they do note iterate past an unprocessed IM operations or a region for which pre-assignments have not been issued. This check is performed regardless of whether the IM operation falls within the thread's partition.

Processing of IM operations is done by the same "continuous" thread as with red operations due to its uniformity requirement.

4.2.7 Latency Compensation and Delayed Serialization of IM operations. As explained in section 3, we assume a geo-distributed system where operations experience non-trivial delays between being produced and reaching all subscribers. This scenario poses a challenge for supporting IM operations, as they can block the application of all other operations at a subscriber until they are received and applied, regardless of the subscriber's distance from the IM producer. Although it is not possible to fully counter this bottleneck, we can attempt to reduce its impact by serializing IM operations later in the log. Placing an IM operation in a distant slot while broadcasting it immediately to subscribers allows them to receive the operation in advance, thereby avoiding subscriber halting due to a missing immediate operation.

Figure 6 illustrates the problem and the proposed technique. It shows the pre-assigned slots for two clients, each positioned in different geographical regions. One client only produces normal (non-IM) operations, while the other only produces IM operations (colored as purple). The one-way network delay between regions is $100ms$. The time displayed above the log represents the calculated instant for an operation to be produced (based on the rate).

In Ⓐ we can see that while both operations 1 and 2 are produced at the same time. However, due to the network delay between regions, operation 1 will only arrive at region B at $t = 100$, blocking all blue and red operations from executing between $t = 0$ and $t = 99$. If, however, the system is aware of the maximum network latency, it can delay the serialization of IM operations by that amount so as to not block the blue operations, as illustrated in part Ⓑ, resulting in the serialization illustrated in part Ⓒ. Clients are agnostic to this reordering and will keep sending the operation at time $t = 0$ but because of the reordering client A will fill the slot serialized at $t = 100$, allowing it to reach the subscriber "on time" and without blockages.

RB-Log implements this compensation scheme by extending the original ZipLog slot distribution algorithm with the expected time for each operation to be produced. It then adds the maximum network latency to IM operations and performs a new reordering. There is also a backlog for entries whose time to send was greater than the epoch size. These entries are placed in the corresponding slot when the next epoch's pre-assignments arrive. The maximum latency value is a per-shard figure and represents the expected time it would take for an operation produced in a given shard to be propagated to all subscribers across all regions. This implies that, as different shards are closer or further for each other, their max latencies will differ and a IM operation originated at a certain region can be serialized much sooner than one from a more distant region.

This optimization, while preventing a subscriber from halting the processing of operations due to in-transit IM operation, will come at the cost of increase latency for IM operations. We argue that such tradeoff is worthwhile, assuming that these operations are not common. This method does not fully prevent halting due to immediate operations, since these still require a continuously processed log up to their sequence number. It does, however, eliminate halting for immediate junk entries as, upon receiving them, the subscriber can immediately "process" them and apply blue operations with higher GSNs.

5 Evaluation

In this section, we present results from a series of experiments we performed to evaluate RB-Log. We will use the original ZipLog [6] implementation as a baseline.

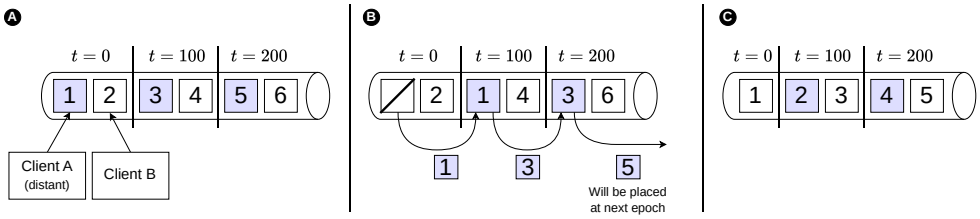

Fig. 6. Example IM operation reorder.

5.1 Evaluation goals

The primary goal of this evaluation is to determine whether support for hybrid consistency improves application-level performance, particularly by reducing latency for Blue operations and increasing subscriber throughput. Specifically, we measure the processing and latency gains of our implementation in a geo-distributed context. Furthermore, we examine how various factors affect system performance, including the proportion of Red, Blue, and IM operations, network instability, and the latency compensation scheme. Finally, we analyze the results to provide explanations for the observed behavior of the system.

5.2 Experimental Setup

For the experiments, we deployed our prototype in three docker containers, each running in a separate physical machine in the same cluster. Each machine is equipped with two Intel Xeon Gold 5320, an Intel 10G X550T NIC and Ubuntu 22.04.4 LTS. The average RTT measured between machines was $0.3ms$.

The performance of ZipLog and RB-Log is affected by the ability of producers to send operations (or NO-OPs) at their specified rate and on time. Clock skew can, thus, negatively affect the latency of operations. To limit clock skew, we synchronized the system clocks of the machines using the Precision Time Protocol (PTP). We do not consider this synchronization to be a limiting factor of our system, as thanks to advancements in synchronization algorithms [8], microsecond-level clock synchronization is now easily achievable across datacenters from cloud providers such as Google Cloud [7].

Each machine represents a region where the system is operating. It includes a Producer-Subscriber client and a single storage server with its own shard that was not replicated. One machine has the additional responsibility of hosting the ordering server. To simulate the distance between regions, network latency was artificially introduced using the `tc` utility. The latency values were derived from the measured RTTs obtained in the experimental setup of Olissipo [11] and are presented in Table 1.

	Region 1	Region 2	Region 3
Region 1	-	44ms	81ms
Region 2	44ms	-	35ms
Region 3	81ms	35ms	-

Table 1. Latency between regions.

All experiments were run for one minute and repeated three times. The values shown in the plots correspond to the averages across all runs. In each run, clients used different seeds to generate the workload. The subscribers were configured with 6 poll threads, 6 processing blue threads and one uniform thread. For all experiments, the initial 20% of operations were discarded as warm-up, and the final 20% as system cool-down.

All presented figures use the following notation: we define end-to-end latency (E2E) as the time between a "client" sending an operation and when the operation is processed in its local machine. Subscriber waiting time is defined as the time between an operation arriving at a subscriber and it being processed. All figures exclude junk (NO-OP) operations.

5.3 Benefits of Blue Operations

We begin by evaluating the performance gain of blue operations over red ones. For this experiment, we launch three clients at a fixed rate of 1000 IOPS. We then vary the percentage of red operations by randomly assigning each operation as red or blue according to a uniform distribution. As we aim to test the theoretical best performance for Blue operations, the number of explicit dependencies is set to 400. This value is sufficiently high to prevent most operations from reaching the limit and thus introducing fake dependencies. We compare our system to the base ZipLog implementation. Figure 7 compares the median, 90th and 99th-percentile end-to-end latency at the subscriber. Figure 8 shows the median throughput and Figure 9 presents the median length of the fine-grained dependency section.

It can be observed that our prototype surpasses the baseline implementation significantly as long as the percentage of blue operations does not surpass 50%. Then, as the ratio of red operations increase, the performance of both systems will converge: this is to be expected, given that with 100% red operations, all operations need to be executed sequentially and both systems will behave identically.

Introducing Red operations significantly reduces both the number of explicit dependencies and subscriber throughput, by roughly an order of magnitude. At 0% Red, Blue operations can be applied without waiting for the log to fill, which combined with the network latency leads to an almost certain incomplete log and thus a higher number of dependencies. As Red operations are introduced, producers must wait for remote GSNs before appending a new operation, which slows down throughput and has the additional consequence of resulting in fewer dependencies, since subsequent Blue operations will observe a much more "uniform" log.

We further examine the impact of our emulated network delay in the system by plotting the median latency per color and region as

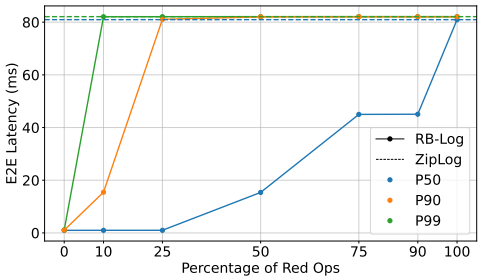

Fig. 7. Median operation latency per subscriber (all regions combined).

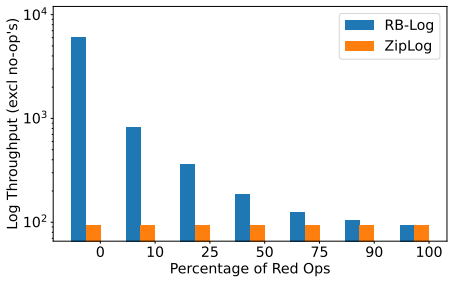

Fig. 8. Median subscriber throughput (mean of all regions).

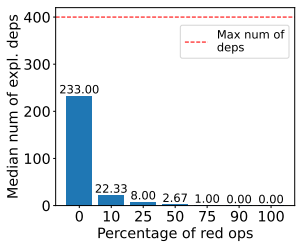

Fig. 9. Number of blue fine-grained dependencies.

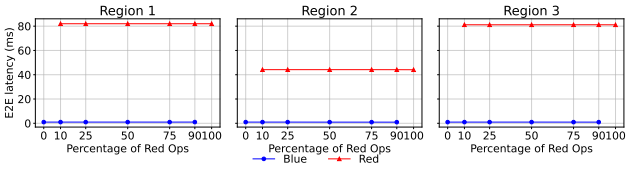

Fig. 10. Median operation latency for Red/Blue per region and per operation type.

shown in Figure 10. We can observe blue operations have the same value across regions while the red ones have the same delay at the maximum network latency between the regions (around 80*ms* for region A and C and 40*ms* for region B). This is to be expected, as blue operations only depend on the causal past of that particular client and thus can be applied immediately at the client who produced them. Red operations, however, must wait for all past operations which, assuming the system is synchronized, means waiting for the operations that were produced at that instant in the other regions. Thus the E2E latency is equal to the network delay of the furthest region.

5.4 Performance of Immediate Operations

Having analyzed the performance of Red/Blue operations, we now turn our attention to the Immediate Operations. As described in section 4, IM operations require synchronization among all operation types. Because of that, a IM slot effectively halts processing until it is applied. To prevent such stalls, we serialize these operations further in the future, ensuring they have time to propagate to all regions without hating processing. This allows the system to incur the latency penalty only when a real IM operation is issued.

To test the impact of IM operations, we use the same setup as in the previous section, but activate the IM stub at a rate of 1000 IOPS, matching the Red/Blue stub. We then vary the proportion of IM/Blue operations (following a uniform distribution) with and without latency compensation. Figure 11 shows the median E2E latency per region and per operation type with latency compensation and without latency compensation.

From region 1, we observe that latency compensation keeps blue operations considerably faster than the baseline when the percentage of IM operations does not exceed 25%. Meanwhile, immediate operations exhibit twice the latency compared to the baseline. As discussed in Section 4.2.7, the compensation mechanism trades off IM operation latency to maintain the execution rate of Blue operations. When it is disabled, IM operations are serialized immediately and must be propagated before any pending Blue operations can be processed. Similarly, because IM operations depend on a continuous log that also requires all Blue operations to be propagated, both conditions result in equivalent latencies for Blue and IM operations.

We also observe that the median latency of blue operations rises as the percentage of IM operations increase. This occurs because a higher amount of IM operations results in more blue operations unable to be processed immediately. The latency for IM operations is fixed at two times the network latency. This is to be expected as IM operations are serialized with a stride equivalent to the max network latency. Furthermore, while IM operations are scheduled in the future, the blue operations for the same epoch are not so they must propagate before we can have the continuous log which adds the extra max network waiting time to the E2E latency.

6 Impact of Network Latency Variance

Finally, we evaluate the performance of our system when it faces unstable networks, with high latency variance. Network instability negatively impacts the system, as delayed packets increase waiting times for Red and IM operations that depend on a complete log. Moreover, the network compensation scheme assumes a relatively stable network delay; if IM operations are excessively delayed, subscribers may reach an empty IM slot and halt processing of subsequent operations.

For this, we add a linearly-distributed jitter with various maximum values to the Linux traffic controller and vary the ratio of IM/Blue and Red/Blue operations. Figure 12 shows, for each jitter value, the median latency per operation type.

It can be seen that as the jitter value increases, so does the latency of Red and IM operations. That is to be expected, as they require a continuous log as a condition for being processed. Blue operations, however, seem to be affected by the jitter when there are IM operations while remaining unaffected with Red operations. This is also to be expected, as we mentioned earlier, once an IM operation takes longer to be propagated than the calculated delay, the subscriber is unable to make progress on the log preventing blue operations

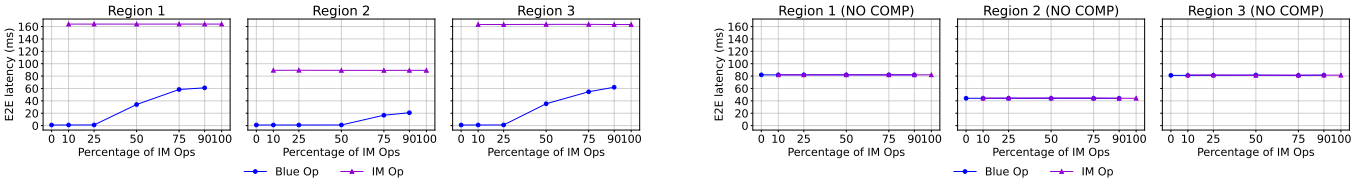

Fig. 11. Median E2E latency for IM/Blue per region and per operation type with latency compensation (left) and without (right).

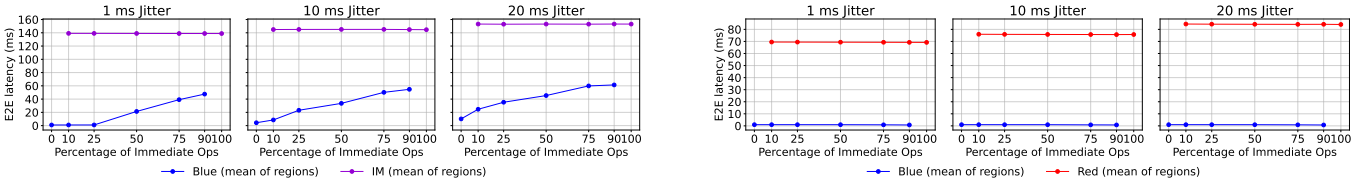

Fig. 12. Median E2E latency for IM/Blue (left) and Red/Blue (right) operations for different jitter values.

from being process until the operation arrives. In the Red/Blue scenario, such halting will never occur, making blue operations always allowed to be processed. Additionally, since end-to-end latency is measured in the shard where the operation is produced, all causal dependencies are already satisfied, allowing Blue operations to be applied immediately.

7 Conclusion

With this work, we reviewed some state-of-the-art systems supporting multiple consistency levels and shared-log architectures. We then presented mechanisms for extending an existing shared log with multi-consistency support, together with its prototype implementation. In particular, we proposed a novel approach for supporting immediate operations without requiring expensive coordination. Finally, we conducted an evaluation demonstrating that a multi-consistency shared log can significantly improve performance compared to a totally ordered one.

In the current prototype a subscriber only executes a Red operation after execution all previous operations (Red, Blue or IM). This is stronger that strictly required by the Red-Blue model, where a Red operation only needs to wait for previous Red operations. Implementing Red operations in a more efficient manner is left for future work.

Acknowledgments

I want to thank Rafael Soares for his guidance and productive discussions. 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).

References

[1] Apache kafka. https://kafka.apache.org/, Accessed: 10/01/2025.
[2] Logdevice. https://logdevice.io/, Accessed: 10/01/2025.
[3] Zeromq socket api. https://zeromq.org/socket-api/, Accessed: 6/10/2025.
[4] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., and Davis, J. D. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, Apr. 2012), USENIX Association, pp. 1–14.
[5] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, Association for Computing Machinery.
[6] Ding, C. *Building A Scalable Shared Log*. Phd thesis, Cornell University, Ithaca (NY), USA, Dec. 2020. Available at https://ecommons.cornell.edu/server/api/core/bitstreams/df90235b-fbd8-4d5e-9313-dd53f3b37cee/content.
[7] Geng, J., Mu, S., Sivaraman, A., and Prabhakar, B. Tiga: Accelerating geo-distributed transactions with synchronized clocks. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025* (2025).
[8] Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., and Vahdat, A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018), USENIX Association.
[9] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.
[10] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making Geo-Replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 265–278.
[11] Li, C., Preguiça, N., and Rodrigues, R. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 359–372.
[12] Lockerman, J., Faleiro, J. M., Kim, J., Sankaran, S., Abadi, D. J., Aspnes, J., Sen, S., and Balakrishnan, M. The FuzzyLog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 357–372.
[13] Renesse, R. V., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)* (San Francisco, CA, Dec. 2004), USENIX Association.