Shared Logs With Support for Multi-Consistency

Francisco José Santos de Carvalho

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

October 2025

Acknowledgments

Firstly, I would like to express my sincere gratitude to my thesis advisor, Prof. Luís Rodrigues, for the opportunity of working under his supervision and for the guidance and support that has made this thesis possible. I would additionally like to give a special thanks to Rafael Soares for all the patience, dedication and productive discussions we had and whose contribution was indispensable for the completion of this work.

I also want to thank my parents, my sister, and family at large for all their encouragement, love, and patience throughout my studies.

To Adriana, thank you for your endless love and for always believing in me.

Last but not least, to all my friends and colleagues. I especially want to thank João, José, Rita and Daniel who always had my back and whose presence made this journey that much more pleasant.

To each and every one of you – Thank you.

Abstract

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 dissertation 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.

Keywords

Distributed Systems; Shared Logs; Consistency; Multi-Consistency

Resumo

Históricos partilhados são um componente essencial de sistemas distribuídos, sendo tipicamente utilizados para registar uma sequência totalmente ordenada de eventos e/ou comandos que precisam de ser (ou já foram) executados no sistema. O processamento de eventos segundo a ordem total definida pelo histórico simplifica o suporte a modelos de coerência forte na aplicação, como linearizabilidade ou serializabilidade. No entanto, essa abordagem pode introduzir problemas de desempenho, especialmente quando o sistema sofre atrasos no preenchimento das entradas do histórico. Uma forma de mitigar esse problema é explorar a semântica da aplicação, permitindo, por exemplo, que operações comutativas sejam processadas em paralelo e/ou que sejam executadas mesmo quando o histórico está ainda incompleto. Nesta dissertação, mostramos como modelos de coerência híbrida podem ser incorporados em históricos partilhados. Apresentamos também o RB-Log, um histórico partilhado que suporta o modelo de coerência Vermelho-Azul com operações imediatas. Avaliámos experimentalmente o RB-Log e mostramos que é possível reduzir de forma significativa (até duas ordens de grandeza) a latência do processamento das operações azuis, desde que a percentagem das operações vermelhas não exceda os 50%.

Palavras Chave

Sistemas Distribuídos; Históricos Partilhados; Coerência; Multi-coerência.

Contents

List of Figures

List of Tables

Acronyms

DAG	Directed Acyclic Graph
E2E	End-to-End
FPGA	Field-Programmable Gate Array
GSN	Global Sequence Number
IM	Immediate Operations
NIC	Network Interface Card
OPS	Operations per second
PTP	Precision Time Protocol
RTT	Round Trip Time
SSD	Solid-State Drive

1 Introduction

Contents

1.1 Motivation

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 [3–5], 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 [6]. 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 dissertation is based on the observation that none of the existing state-of-the-art multi-consistency systems leverage 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.

1.2 Contributions

This dissertation provides the following contributions:

- A survey on how previous systems implement multi-consistency support;
- A set of techniques to support hybrid-consistency in totally ordered logs.

1.3 Results

This thesis produced the following results:

- A novel multi-consistent shared log prototype, named RB-Log, supporting strong and weak consistency, that has been implemented by extending the state-of-the-art ZipLog [7] system;
- Experimental evaluation of the prototype, covering the performance of all operation types in a simulated geo-distributed environment.

1.4 Research History

This work is part of an ongoing research project at INESC-ID that aims at building scalable and efficient shared logs. This work benefited from the comments of Rafael Soares on the design, implementation, and evaluation of the prototype. Our implementation is based on the shared logs proposed in the Phd Thesis of C. Ding [7].

Part of this work was presented as a Poster at the “Décimo sexto Simpósio de Informática, Inforum 2025, Évora, September 2025”.

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).

1.5 Organization of the Document

The rest of this document is organized as follows: In Chapter 2 we start by presenting the necessary background information for the understating of the systems presented in the same chapter. In Chapter 3, we provide an overview of the proposed solution, along with its architecture and design decisions. Chapter 4 details the implementation of the described architecture. In Chapter 5 we present the results of the experimental evaluation; Chapter 6 concludes the dissertation and provides directions for future work.

2 Background

Contents

In this chapter, we present the background necessary to understand the following chapters, along with a review of related work.

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. Furthermore, as the log is just an abstraction, there are various ways of storing and replicating the entries according to the needs of the underlying application. Most systems use a combination of data sharding and data replication. Sharding partitions the entry-space into various storage servers while replication means making a full or partial copy of all the entries, either to provide fault-tolerance and/or to geo-distribute the log and thus provide a faster service to clients.

Beyond appending and reading positions, systems implementing shared logs often provide the following additional operations:

1. *subscribe*, in which the log will send any newly appended operation to the subscriber client, reducing the number of round-trip messages to the log. This is useful in certain cases such as in the state machine replication model, where all clients are performing the same appended operations and thus must monitor any newly filled positions.

2. *trim*, which allows a client to request an entry, or range of entries to be discarded once the system can guarantee they are no longer needed. This functionality is important as the log is an ever-growing structure, thus the ability to trim old entries contributes to a more effective usage of storage resources.

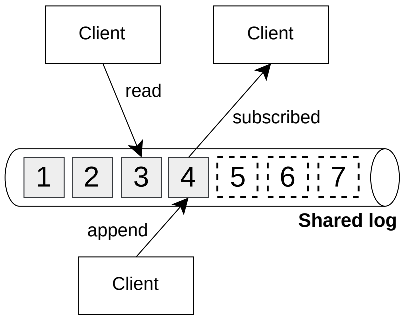

Figure 2.1: Shared log.

Figure 2.1 illustrates a shared log system, with one client reading the position $\boxed{3}$, one client successfully appending an operation at the tail that then gets propagated to a subscribed client.

While the shared log abstraction provides a convenient mechanism for establishing a total order, concurrent append operations introduce contention, as multiple clients may attempt to append at the same log position (the tail). Because only one operation can succeed, the abort rate tends to be high. This problem is addressed in the existing shared logs systems in two main ways:

- By introducing a position reservation system issued by a single entity, named a sequencer;

- By establishing a total order of operations after said operations have already been replicated.

We will provide more details about these methods in the Related Work (Section 2.3).

2.2 Consistency Models

As previously introduced, geo-replication allows the same data item to be placed in various regions to reduce its access time, amongst other advantages such as fault tolerance. However, as data propagation is not instantaneous, different regions may temporarily observe stale copies of data which, if not accounted for, can result in application errors or lost data (if the system is configured to override concurrent appends only one will become visible). It is therefore important to reason about what safety guarantees does the system impose on replicated data. These safety properties are called consistency models and range from just requiring consistent reads in the same process (with "read your writes" consistency) to guaranteeing a system-wide total order of operations consistent with the real time (with linearizability).

Consistency is an integral part of this work and distributed systems in general. As such, this section presents some of the consistency models used by the systems in the related work section. More models can be found in [8, 9] (the consistency model descriptions presented below were partially adapted from those sources).

Consistency models can be divided into two groups: single-object and multi-object (more commonly known as transactions). Informally, a consistency model can also be labeled as strong, if the model imposes a total order or weak if it's more permissive. One example of weak consistency is causal consistency, in which causally related operations are seen by all processes in the same order. Concurrent operations, being causally independent, may be applied in different orders across replicas. This behavior is, however, prohibited under strong consistency models such as linearizability, which impose a total order on all operations, ensuring that all replicas apply them in exactly the same order.

2.2.1 Single-object Consistency Models

We start by defining four weekly consistency models. These are collectively known as "session guarantees" [9]. A *session* is defined as a sequence of read and write operations. Session guarantees will govern the ordering of reads and writes within an individual session.

Read your writes ensures that, for the same process, every read executed after a write must reflect the changes of that write. This restriction only applies to operations made by the same process.

Monotonic reads guarantees that the same process can not make two sequential reads and, in the second one, obtain a state that does not incorporate the changes observed in the first read.

Monotonic writes ensures that sequential writes from the same process must be observed by all processes in that same order.

Writes Follow Reads ensures that all previously observed reads in a process must be observed in any subsequent writes.

Building on the session guarantees, the following models enforce increasingly stronger forms of consistency:

Causal Consistency guarantees Lamport's *happens-before* relation (represented as $\rightarrow$ and formally defined in [10]). Under this model, causally-related operations should appear in the same causal order at every replica. Specifically, if operation a happened before operation b (denoted as $a \rightarrow b$) then b can only be applied after a. If the operations are concurrent (denoted as $a \mid\mid b$), no happens-before relation can be established. As such, no ordering guarantees are enforced. This model establishes a partial order of operations, allowing replicas to apply them in any order that preserves causality.

Causal+ Consistency or "causal consistency with convergent conflict handling" [11] is an extension of causal consistency that ensures that if there are concurrent updates in the same item and different replicas, eventually that conflict will be resolved and the state of the replicas will converge.

Linearizability [12] is the strongest single-object consistency model. Under linearizability, operations execute in a total order that is consistent with the real time. In practice, this model ensures the system behavior is equivalent to that of a non-distributed and non-concurrent system.

There are other models that are relevant to our work since they allow applications to combine multiple levels of consistency, namely PoR [3] and RedBlue [4] consistency. These models will be presented in more detail, alongside their systems, in the Related work section 2.3.

2.2.2 Transactional/Isolation Models

Isolation Models govern the order of multi-object operations (transactions). Such operations can be understood as sequences of read and/or write operations that are executed atomically as a single unit.

The isolation model therefore specifies safety conditions for the correct ordering of transactions within the system. We now present some of these models.

Monotonic Atomic View ensures that if some effects of a transaction T_1 are observed by a transaction T_2, then all operations within T_2 will observe all effects of T_1.

Transactional Causal Consistency, introduced in Cure [13], extends the Causal+ consistency model to transactions. Under this model, all transactions execute their operations from the same causally consistent snapshot, which includes all the operations in the issuing client's causal past. As it derives from causal+, this model assumes that it is possible to merge together two concurrent transactions (specifically, using CRDTs [14]) so as to guarantee state convergence.

Serializability ensures that the concurrent execution of multiple transactions is equivalent to that of some serial execution order. It requires a total order of execution across all transactions. This model does not, however, guarantee that the order will be coherent with the real time order.

Strict Serializability guarantees the same properties as with Serializable with the added restriction that the order of operation must be coherent with the real time order.

2.2.3 Multi-Consistency

Having introduced various consistency models, we now broadly discuss how and why multi-consistency is supported in some systems.

As noted in Chapter 1, application developers traditionally must choose between strong consistency and high throughput. Systems providing strong consistency are unable to achieve the same performance as their weaker counterparts, as their total order requirements force sites to coordinate the order of execution. Meanwhile, in weakly consistent systems, operations can be executed with little to no coordination, but that also introduces significant drawbacks of stale copies and complex data reconciliation. While some applications are able to tolerate weak consistencies' drawbacks and enjoy the additional performance and scalability, others cannot and are thus forced to pay the performance penalty.

There are, however, cases where only a small subset of operations require strong consistency in order to ensure application correctness. Using the example from RedBlue [4], if a bank application must never allow a negative balance, the withdrawal operation will require synchronization, as concurrent executions may violate such requirement. However, operations such as deposits are safe to be executed under weak consistency, as they will never lead to a negative balance.

Multi-consistency systems aim to provide the best of both worlds to such applications: they ensure strong consistency to operations that require it while allowing the remaining operations to execute under faster weaker consistency. The idea of supporting multi-consistency has been partially implemented in some popular systems like Amazon DynamoDB [15] which, under certain conditions, supports strong consistent reads. There are other systems, such as Gemini [4], that were designed to support multiple levels of consistency, giving developers the flexibility to choose the appropriate consistency type for each operation. Furthermore, there are also systems like Unistore [16] that support multiple levels of consistency in the context of transactions, allowing the client to specify at run-time the level of consistency for each one.

While offering a good balance between consistency and performance, multi-consistency systems impose an additional burden on application developers, who must determine which operations should execute under each available consistency model. Some research has sought to automate this process, with SIEVE [17] being one such system. These techniques, however, are beyond the scope of this work.

In the next section (2.3.1), we dive deeper on multi-consistency systems.

2.3 Related Work

In this section, we present some of the state-of-the-art systems that offer multiple consistency levels and shared logs. For each system, we provide an overview followed by some relevant implementation details. Each solution will then be compared.

2.3.1 Systems Supporting Multiple Levels of Consistency

Lazy Replication [18] describes a system for replicating state across multiple servers. Clients issue operations against a single replica, which are then propagated 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 operations 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 consistency.

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 an additional system-wide synchronization using a 3-phase protocol.

- *Pre-Prepare phase* where the primary will ask each replica for their log and timestamp in order to register the dependencies of the immediate operation. When a replica receives this message, it can no longer respond to clients as it is not sure whether the operation committed or aborted.

- *Prepare phase* starts when the primary receives acknowledgements from all the nodes. It increments the forced operation counter and assigns it to the immediate operation. It then sends the update to all replicas. As with replica nodes, after this phase, the primary cannot guarantee if the operation succeeded and as such cannot respond to new requests until commit.

- *Commit phase* is started when a majority (including the primary) acknowledged the update. The primary commits the operation and informs the client. As with forced operations, the backup replicas will know of the commit via gossip, but they will only be able to restart responding to requests after receiving it (contrary to forced updates that do not impact normal operations).

The process of a view change during the execution of a immediate operation works similarly as with forced operations. Every operation that has been prepared will be ported to the new view, where phase 2 will be repeated. If the operation has not been prepared, it will abort.

Gemini [4] 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.

Olisipo [3] is the coordination service for geo-distributed applications that implements the partial order restrictions consistency model (PoR consistency).

PoR extends the RedBlue consistency model by allowing finer-grained total order requirements between pairs of red operations, rather than enforcing a uniform total order among **all** red operations as in the original model. This enables more precise specification of ordering constraints between red operations. RedBlue requires a total order between all red operations, which is safe but often overly restrictive when operations only conflict with a small subset of others. PoR addresses this by removing the Red/Blue distinction and allowing each operation to specify the set of operations it must synchronize with (its conflict operations) enabling more precise ordering constraints. Operations that do not have conflicts correspond to the original Blue operations and can be executed concurrently, whilst operations with conflicts will synchronize and define a total order of execution between conflicting operations. The original Red operations can thus be defined as operations whose conflicts are the set of all other operations that would execute under strong consistency. As with RedBlue consistency, to ensure state convergence, operations without conflicts are required to be globally commutative. Any operations that do not have this property must execute under strong consistency and, consequently, are required to list any operations that cause non-commutativity as their conflicting operations. Hence, PoR provides more flexibility for application developers when developing over multiple levels of consistency, requiring only the identification, for each operation type, of the minimal set of restrictions that make the system behave correctly.

This model is implemented in Olisipo. Unlike the systems presented earlier, Olisipo is not a storage system. It is a coordination service designed to sit on top of an existing distributed storage system, ordering operations using the PoR model and a coordination strategy. This coordination service is composed of multiple *agents* that order append requests from their local data center. Coordination between two conflicting operations can be achieved through multiple means (Paxos, Distributed locking, amongst others). For flexibility, Olisipo provides two possible coordination protocols for each pair of conflicting operations:

- **Symmetric**, where both operations are required to coordinate. This is implemented with a counter service, totally ordered through Paxos. For each conflict pair, two counters are stored, representing the number of operations appended of each type. Whenever a new operation in the conflict pair is appended, local agents cross-check their counters with the global service to check whether they have applied all prior operations, if so the operation can be safely applied.

- **Asymmetric**, which assumes one operation will be executed more often than the other. It allows the most frequent operation to be executed by default without any coordination, imposing the cost of coordination upon the less frequent one. This is implemented using a distributed barrier system, analogous to the immediate operation of Lazy Replication. When a non-frequent operation is submitted to a replica (the primary), it contacts all other replicas, which collectively pause the processing of the conflicting operation. The append operation can then be completed, after which the appends of the other operations resume.

The choice of each protocol to use is made dynamically based on the usage pattern of each operation.

Under the Gemini system, Olisipo replaces the concurrency coordinator (that enforces the RedBlue order).

Unistore [16] is a geo-distributed data store that combines strong and causally consistent transactions, implementing the PoR model. Causal transactions can be applied concurrently and are always seen by the clients in a causally consistent order whilst strong operations that conflict with each other must be applied in some serial order.

The system is composed of multiple geo-distributed data centers, each with multiple servers that store data items. Each data center contains a complete (possibly stale) copy of all data items. Additionally, each update to a data item is represented as a log of updates that can be replayed to reconstruct its state.

Causal transactions leverage the Transactional Causal Consistency execution model of Cure [13], which served as the basis for this system. Causal transactions are executed in the client's local datacenter and are forwarded in the background to remote datacenters. Transactions are executed against a causally consistent snapshot, which includes all transactions on the client's causal past. Client's causal dependencies are encoded as vector and version clocks and are kept alongside transactions as metadata. Remote datacenters may only apply a causal transaction once all its causal dependencies have been met. Furthermore, a transaction's updates may only be made visible once the transaction has been made uniform (i.e. it has been made durable at $f + 1$ data centers). Similarly to Cure's [13] stabilization protocol, replicas periodically exchange these vectors to determine which operations are uniform and whether any forwarding is needed.

Strongly consistent transactions are required when two conflicting operations access the same data item. These transactions require synchronization between replicas, which is accomplished through a certification service. Strong transactions are executed optimistically in the local data center and subsequently submitted for global certification. While causal operations can be committed to the log immediately after execution, a strong operation is only committed to the log after being certified. The certification service ensures that no conflicting operations are executed concurrently on the same data item. The certification service runs a combination of two-phase commit and paxos to ensure the correct ordering of operations. All dependencies of a strong operation must be uniform prior to sending the transaction for certification. That is because if a data center fails and never sends a required dependency to a replica, the strong transaction cannot be certified. Since the strong transaction is not applied, all subsequent operations that conflict with it are also blocked. Forcing dependencies of strong transactions to be uniform guarantees that every dependency will eventually be forwarded by at least one correct data center.

Ensuring that all dependencies of a strong transaction are uniform is costly, as it requires waiting for their execution at all replicas. To prevent this, local replicas only expose remote operations to their local clients when they have been made uniform. Strong transactions then locally execute in a snapshot with dependencies that are already uniform. With this optimization, a strong operation only waits for locally-produced causal operations to become uniform. This is necessary because locally produced operations must be made available to clients immediately after being committed to preserve the read-your-writes semantic.

2.3.2 Systems Based on Shared Logs

Corfu [19] 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 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 Field-Programmable Gate Arrays (FPGAs) attached to Solid-State Drives (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 [20].

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 entry'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.

Scalog [21] is a shared log system that attempts to mitigate some of the drawbacks of Corfu, namely the complete halting of the system during reconfigurations, the sequencer as a bottleneck and the need to run custom hardware.

This is achieved by decoupling operation ordering from the append process. Clients send entries to their preferred shard, where they are stored on one of the shard's servers. Each server runs a replication scheme for its entire shard, following the Primary-Backup strategy. When an operation first arrives at a server, it is placed in its primary section. From there, operations are forwarded in FIFO order to all storage servers within the shard, which act as backups. Then, from time to time, all servers send their operations to a group of replicated nodes running Paxos (collectively called the *ordering layer*). These nodes check whether an operation has been fully replicated within the shard and assign it a position in the total order. Once the log position is given, the server can return to the client. In Scalog, the state of replication is captured with vector clocks, where every storage server registers their last appended (primary) entry and their last received (backup) entry. Representing operations with a single number is possible because replication is performed in FIFO order. These scalars, known as the *local cut*, are sent by each server to the ordering layer, where they are aggregated into a new vector clock representing the durable operations of the entire system, referred to as the *global cut*. The sequence of global cuts is then used to establish a total order. By subtracting two consecutive global cuts, the system identifies the newly durable operations at each shard. Intuitively, operations in a later global cut are ordered after those in the previous one. Within a global cut, entries are sorted by shard and server IDs, establishing both inter- and intra-cut ordering and resulting in a total order across all operations.

In Scalog, all operations execute under linearizability. Reads can be performed at any server within a shard but, before any value is returned, the storage server must first check whether the server's local data is sufficiently up-to-date with the clients last observed cut. This is enforced by checking whether the log sequence number supplied by the client is smaller than the last operation observed by the server. Additionally, a client can *subscribe* to the log where new globally ordered operations are returned by random servers at each shard.

Reconfigurations are used to add or finalize shards (making them read-only), either to recover from failures or to adjust capacity. Unlike Corfu, Scalog allows shards unaffected by failures to continue operating. This is possible because Scalog orders entries only after they are appended and does not maintain a distributed mapping of log positions. A server failure is detected as it stops sending *local cuts* to the ordering layer and a reconfiguration process is initiated. The process of reconfiguration can be made in three ways: 1) The entire shard can be finalized and data is read from the other healthy servers; 2) The server can be replaced and data copied from the backups. During this process, writes in the shard are halted; 3) if there are more than $2f + 1$ servers per shard the faulty server can be masked, ensuring immediate availability for reads and writes.

Dapple (FuzzyLog) [5] is a distributed shared log with geo-distributed replication built around the FuzzyLog abstraction, a partially ordered shared log. FuzzyLog does not impose a system-wide total order, which is costly and may be impossible in cases of network partitions. Instead, it relies on multiple distributed logs, organized into a Directed Acyclic Graph (DAG) where each edge captures a causal dependency between two operations. All operations targeting the same logical shard share the same *color*, and operations from the same site within that color form a totally ordered *chain*. The full DAG is replicated at all sites in the following way: for each color, there is a (totally ordered) local chain with the operations appended at that site and (lazily synchronized) remote chains with operations that originated on remote sites. Such loose synchronization enables updates of the same color to be performed concurrently across regions and with no conflicts, as the updates are appended to different chains. If there is only one color and one region, FuzzyLog behaves as a strongly consistent shared log.

Dapple is a system that implements these abstractions on top of a collection of geo-distributed servers (*chainServers*). Colors are stored in a single partition and replicated using chain replication [20].

A client appends data to the tail of its local chain in the color of the selected shard, obtaining a coarse-grained lease from the corresponding server to do so. The lease ensures fault tolerance and requires the operation to complete within its duration. The operation also contains links to the last node seen by the client in each remote chain for that color (reflecting its causal past). When a client joins any given site, it must first synchronize its *view* with the shard to ensure it contains all of the client's causal history. In this process, the client may have to block until remote changes arrive and are executed at the local site. Operations can modify multiple colors simultaneously. When this occurs, Dapple uses a modified version of Skeen's algorithm [22] to synchronize the insertion across the affected chains.

The FuzzyLog abstraction is used in the Dapple system to implement multiple applications. For example, *LogMap* is a system that uses a single color in one region, running a totally ordered shared log and providing linearizability. In *atomicMap* each server stores a color that is being constantly synchronized with the remote servers with the same color. It uses the serializability consistency model and can also support multicolor appends where an operation is appended atomically at two colors.

CAPmap uses linearizability under optimal conditions, downgrading to causal+ consistency in case of network partitions. This is accomplished by routing all append operations through a pre-defined single *primary region* server who thus establishes a total order. If the primary server becomes unreachable, the servers begin appending operations to their local chains and thus lowering the consistency to causal+.

2.3.3 Comparison

In this section we present a brief comparison of the systems presented thus far. As we've done in the previous section, we will distinguish between shared logs and systems with multiple consistency levels. As Dapple is able to guarantee various consistency levels depending on the specific implementation, we decided to compare it with other shared logs as it is the main feature of the system. We also include our solution for a shared log with multi-consistency support, RB-Log, whose implementation is described in Chapter 4.

All these systems build upon their previous iterations. LazyReplication laid the groundwork for Red-

Blue consistency. PoR can be seen as a generalization of RedBlue, and Unistore uses it to provide liveness and multi-consistency in a transactional system. The same can be said for the shared logs systems, where Corfu introduced the log abstractions whose subsequent work of Scalog and FuzzyLog is based on.

Shared Logs

System	Consistency model	Data Replication	Log ordering
Corfu	Linearizability	Chain replication	Sequencer assigns positions
Scalog	Linearizability	Primary-Backup	Paxos-based ordering layer
Dapple (AtomicMap)	Linearizability	Chain replication	Modified Skeen algorithm
Dapple (CAPmap)	Linearizability (base case); causal+ (unavailable primary)	Chain replication	Total order when primary in use; causal order otherwise
RB-Log	RedBlue + Immediate	Replication of ops within the shard	Pre-ordering of operations by a sequencer

Table 2.1: Comparison of properties in shared logs.

Table 2.1 provides an overview of the points we will discuss for each system.

Consistency Model: Both Corfu and Scalog provide linearizable operations. In FuzzyLog, the level of consistency depends on the specific implementation used. LogMap assumes a single region and color, with all entries executing under linearizability. CapMap provides linearizability if the local server can reach the primary (and proxy the requests through it). If the primary becomes unreachable, it downgrades to causal+ consistency for the new operations until connection is restored.

Data Replication: In Corfu, data is replicated in multiple servers using a model based on chain replication. Furthermore, a failure requires a reconfiguration to change the projection, which halts the entire system. In Scalog, data is replicated between the shard servers and can be copied in case of a fault. As with Corfu, a faulty server involves a reconfiguration, but the unaffected shards are able to continue appending information and, depending on the policy of the affected shard, may still keep processing new requests (by masking the faulty server). Dapple, which is the system that implements FuzzyLog and their systems, uses chain replication for data replication, similarly to Corfu.

Log Ordering: In Corfu, operations are totally ordered based on the physical position where the client wrote the operation, which is reserved by a centralized sequencer to reduce contention. In Scalog, entries are durably stored by shard servers; a Paxos-based ordering layer subsequently agrees on their order. Dapple totally orders operations upon arrival when they span a single color by appending them to the end of the corresponding chain. For operations that span multiple colors, it employs a variation of Skeen's algorithm [22] to establish a total order. In case of CAPmap, operations are forwarded through a single primary server in order for a total ordered to be established.

Multi-Consistency Systems

System	Consistency model	Metadata	Strong operation impl.	Fault Tolerance
Lazy Replication	Causal, Strong (immediate and forced ops)	Vector Clocks with a strong op counter	Primary-Backup with view changes	Primary-backup (strong ops); replication of ops
Gemini	RedBlue	Vector Clocks with a strong op counter	Token-Passing to increment vector clock	Not implemented (possible impl. of Paxos for tokens)
Olisipo	PoR	Logical clocks	Global logical clock checking or distributed barrier	Paxos-like SMR for counter service
Unistore	PoR	Vector Clocks with a strong op counter	Global certification with 2PC+Paxos	Operation replication; Paxos for certification of strong transactions
RB-Log	RedBlue + Immediate	Log sequence numbers	Total order given by the log	Replication of ops within the shard

Table 2.2: Comparison of properties in multiple consistency systems.

We will compare the multi-consistency systems across 4 categories, summarized in Table 2.2 and described in more detail below.

Consistency Levels: In terms of consistency models provided, all four systems allow applications to execute operations with either strong or weak consistency. Olisipo and Unistore are the systems whose consistency model is the most flexible, as they do not enforce a single total order for all red operations (something that RedBlue and Forced operations impose). In LazyReplication, Immediate operations are a stronger form of Forced (or Red) operations, as they conflict with all other operation types. In the PoR model, such operations could be represented by creating a conflict with all other operations.

Metadata: All systems handle metadata similarly, using either Vector clocks or logical clocks to capture the causal dependencies of operations. Olisipo is a particular case, as it does not track causal dependencies directly. Serving as a coordination service for PoR consistency, it operates on top of a system like Gemini, handling ordering for operations with conflicts. Causal dependencies for blue operations are therefore managed by Gemini.

Implementation of Strong Operations: Each of the storage systems implements strong operations differently, although they all track the total order using a dedicated "strong entry" in the vector clock. Lazy Replication uses a Primary-Backup scheme where the primary server proposes an increment in the strong counter to the backups, which is then certified once a sub-majority (50%) is archived. In Gemini, a token is passed around the sites, allowing whoever possesses it to increment the counter. In Unistore, strong operations are globally certified using a combination of Paxos and two-phase commit. Olisipo has two algorithms for processing conflicting operations, using Paxos or a distributed barrier.

Fault Tolerance: In Lazy Replication, there is fault tolerance for data as an operation will only be applied after all their causal dependencies (operations are eventually fully replicated in all servers), and for strong operations appends, where a failure in the Primary can be recovered with a view change. Gemini does not explicitly implement fault tolerance although the authors propose a number of mechanisms, such as using Paxos instead of token passing and data replication. Olisipo uses Paxos for its global logical clocks and Unistore only applies operations when all their dependencies are present. It also uses a retransmission mechanism to ensure shard failures do not pose a liveness problem.

Summary

This chapter provided the required background on shared logs and consistency models for this dissertation's proposed solution, including a comparison of existing shared-log and multi-consistency systems.

3 A Multi-Consistent Log

Contents

In this chapter, 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 blue 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 operations 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 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" [4], 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:** 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 subsequently 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).

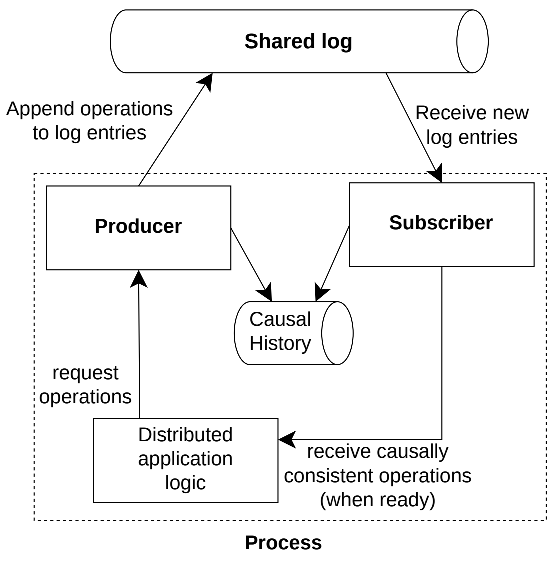
Shared log
Append operations to log entries
Receive new log entries
Producer
Subscriber
Causal History
request operations
Distributed application logic
receive causally consistent operations (when ready)
Process

Figure 3.1: General Architecture.

3.3 General Architecture

Figure 3.1 illustrates a process (or client) composed of a producer and a subscriber. Processes 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. The causal history is expressed in each operation as a timestamp containing the previously seen operations. We discuss several ways of expressing causal history in Section 3.4.

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.

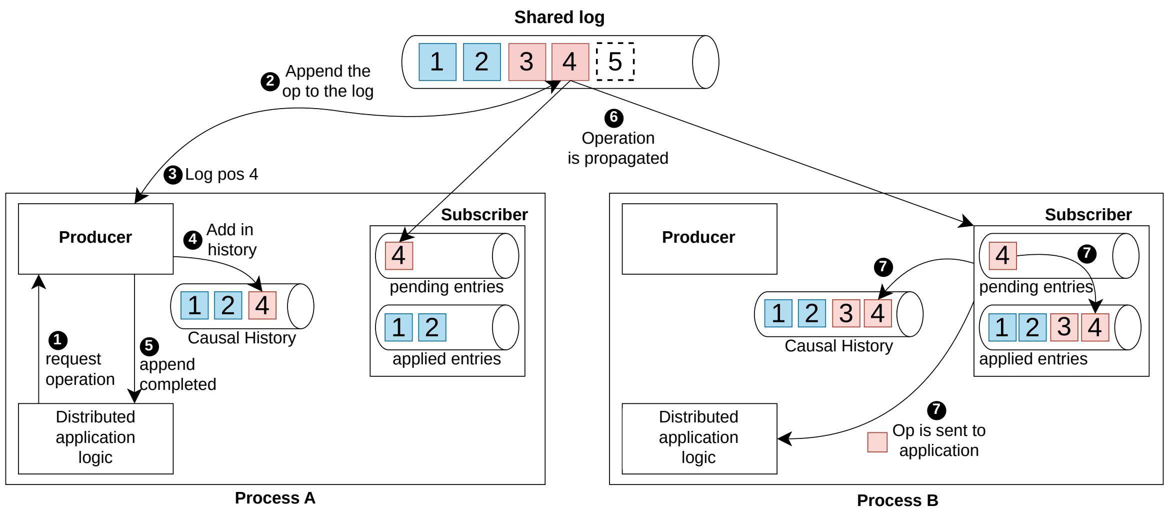

Figure 3.2: Operation with two replicas in different locations.

Figure 3.2 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 $\boxed{4}$ becomes visible at A, but as it is a strong operation, it must wait for operation $\boxed{3}$ 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 $\boxed{4}$ also reaches Process B. In this case, as $\boxed{3}$ is already visible and applied, the operation can be immediately applied.

If operation $\boxed{4}$ was marked immediate instead, Process A would still be unable to apply it. Nevertheless, the behavior would differ for a hypothetical blue operation $\boxed{5}$ arriving at Process A: in that case, it could not be applied until $\boxed{4}$ had been applied, even if the operation had all their causal dependencies present. This is illustrated in Figure 3.3 (the IM operation is represented in purple).

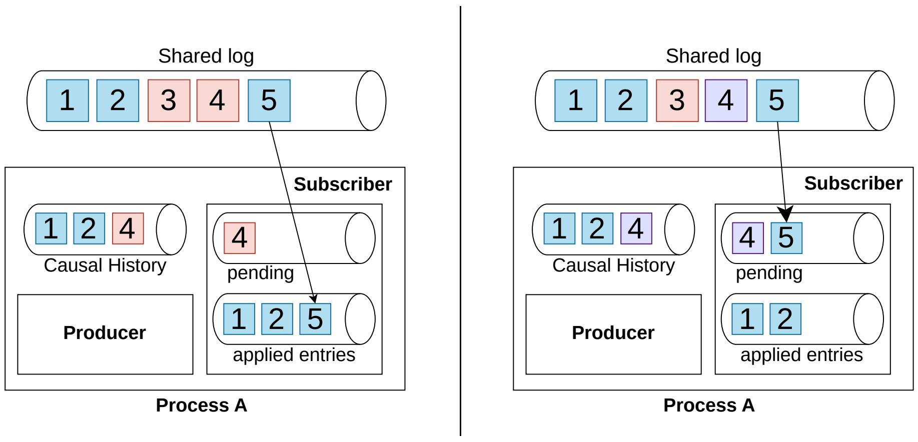

Figure 3.3: Application of a blue operation without an immediate (left) versus with an immediate (right).

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 may require processing, possibly reducing the causal processing advantages if not accounted for. This section presents possible approaches to dependency tracking, as well as their feasibility in our system. Note that, as described above, the causal dependencies could be simply encoded as a list of all operations that have been read or written by a client. Such an implementation would be very expensive, therefore we discuss other more efficient alternatives.

Logical Clocks

The simplest way to track causal dependencies would be to rely on the sequence numbers provided by the shared log. When appending a new entry, clients could attach their highest observed log position (observed either through an append or by applying an entry). The newly-appended operation would then causally depend on all operations with a log position lower or equal than this specified position. This method makes metadata as small as a single scalar and leverages the already existing shared log infrastructure. However, it also introduces fake dependencies between operations, as clients must wait for all positions with lower position to be applied, regardless of the actual observed entries of the appending client. This is because, as presented earlier, blue operation can be played out of log order. If a client has “holes” in its log, the single scalar value is not capable of expressing which operations have been seen. As a result, those missing operations would still be treated as causal dependencies for any new operation.

Vector Clocks

One could try to reduce the number of false dependencies by expressing them using vector clocks. Vector clocks are arrays of integers that maintain a logical clock for each client. All clients store their own vector clock, which reflects the operations it has appended and applied, thereby capturing causal dependencies. Furthermore, by using a single scalar per replica, vector clocks allow for efficient dependency processing. Although vector clocks have been used in previous multi-consistency systems [4, 16, 18], they would require maintaining a view of existing processes in the system, incurring heavy costs on clients membership changes as well as being a bottleneck for systems with a large number of clients. We aim at a solution agnostic to the clients/processes view.

Explicit Dependency Tracking with Top-Most Dependencies

Another way to remove fake dependencies between operations is through the use of explicit dependencies. Explicit dependencies tag each operation with the set of all log indexes previously observed by the client. However, this technique can create an unbounded growth of a clients causal history. To reduce its size, clients constantly remove redundant dependencies by computing the minimum set of dependencies that do not violate causality. This set contains the operations’ top-most dependencies – those not already implied by others. This method can reduce the growing number of dependencies, but it cannot remove them entirely. (This optimization is formally defined in COPS [11]).

Hybrid Dependency Tracking

While the aforementioned optimization reduces the amount of explicit dependencies, there is space for further optimizations. Although one cannot escape from the “holes” in local logs, 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

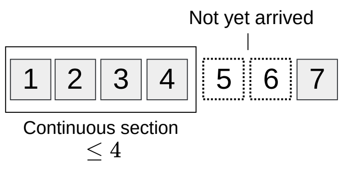

Figure 3.4: Example of a local log with a continuous section and two late operations.

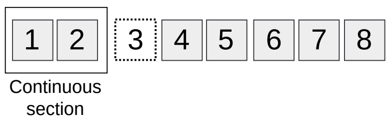

Figure 3.5: Example of how a late arrival of a single operation may make the explicit section much bigger.

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 3.4 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 $\boxed{4}$, and the set denotes the explicit dependencies outside the continuous section. If operation $\boxed{5}$ 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. Consider the execution in Figure 3.5: due to missing operation $\boxed{3}$, 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. Figure 3.6 represents the previous situation under this new model. We define the maximum number of explicit operations λ as 3. As the explicit section has 5 elements, the continuous section is advanced until operation $\boxed{5}$, leaving operations $\boxed{6}\boxed{7}\boxed{8}$ as explicit dependencies. Note that due to this advancement, operation $\boxed{3}$, which was not seen by the client, is now a causal dependency of this operation.

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

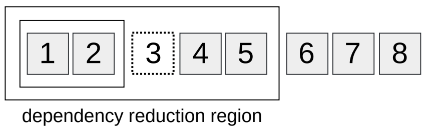

Figure 3.6: Introduction of a fake dependency caused by the dependency ceiling.

Summary

This chapter introduced a novel approach for building a shared log with multi-consistency support. It presented the system architecture and illustrated its operation with an example. Furthermore, it compared various dependency-tracking mechanisms and proposed hybrid dependency tracking as a balanced solution for this system.

4 RB-Log

Contents

This chapter introduces RB-Log, a shared log prototype with multi-consistency support. Instead of building a shared log from scratch, we build 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 [7] 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 [7] 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 expected 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 is implemented in C++ and uses ZeroMQ [23] 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.
- **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.

At startup, each component registers with the ordering server. Producers inform the ordering server the number of operations they intend to append. The ordering server then sets a timer to issue the assignments for the next epoch. Such information is sent to all storage servers one epoch in advance to ensure that all nodes receive the information before the epoch starts. It consists in the set of all operation requests from producers as well as the epoch start time. The ordering server also informs each stub of the epoch start and the number of allocated slots.

Upon receiving the new epoch message, storage servers execute a distribution algorithm based on Bresenham's line algorithm to evenly assign log positions to stubs. As for the stubs, the new epoch message is used to determine how many operations to send and when the epoch begins. Stubs do not calculate their pre-reserved operations themselves, and are instead informed of the log position assigned to each appended operation.

End clients issue operations to the nearest stub, which will queue and issue them according to its expected rate upon epoch start. If no operations are queued, stubs issue special NO-OP operations to ensure the timely filling of the log. Each operation is sent to all replicas within the producer's shard. Upon arrival at a storage server, the operation is placed in the next available pre-assigned entry for that stub. The storage server then returns the sequence number of the filled slot to the stub, referred to as the Global Sequence Number (GSN). The producer waits for at least $f+1$ ack-backs before notifying the application about the successful insert (where f denotes the configured number of failures).

Once a log position is filled on a storage server, the corresponding entry must be disseminated to all subscribers. To achieve this, each storage server employs multiple threads that iterate through filled positions and deliver them to subscribers. Storage servers can also be configured to aggregate these messages into batches prior to transmission.

Upon the arrival of a new operation to the subscriber, one of its “poll” threads places it into a queue, where it waits for at least $f+1$ broadcasts from the storage server replicas before being marked for processing. Once marked, a “processing” thread will apply the operation after all preceding operations have been processed.

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 “polling” 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

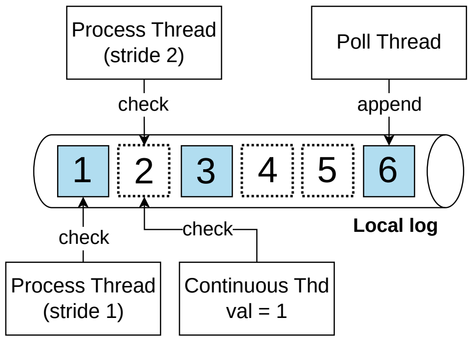

Figure 4.1: Architecture of the subscriber's poll and process threads.

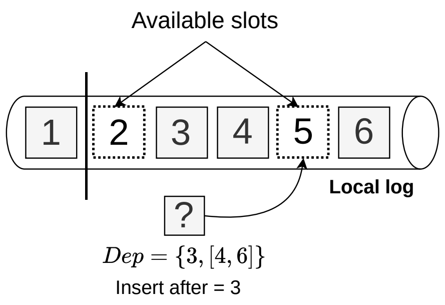

Figure 4.2: Insert_after is required when the system reduces the fine-grained section.

are deemed ready. In RB-Log, the condition for an entry to be ready for application now depends on its operation type: Blue operations may be applied in an order different from the log, provided that their causal past is present, 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.

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 4.1 shows a subscriber's local log and the different threads that interact with it. The Continuous thread is shown halted at the unarrived operation $\boxed{2}$, 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 (with a lower GSN) 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 4.1 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 $\boxed{6}$, which is the highest registered operation.

4.2.4.A 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.

However, due to the way clients are implemented in this system, they do not know in advance the GSN that will be assigned to an appended operation. Combined with the storage servers' logic of assigning operations to the first available slots, this can create a paradoxical situation in which an operation is assigned a lower GSN than its continuous component (i.e., the fake dependency is the operation itself). Such problem can be solved if clients include in every append the `insert_after` field, which instructs the storage server only place the operation after that specified GSN. The available slots in between are then marked as junk. Figure 4.2 illustrates this situation. The operation to be appended to the log must trim one slot, resulting in the timestamp $\{3, \{4, 6\}\}$. However, its lowest available entry is $\boxed{2}$ which would make this operation impossible to apply. By setting the `insert_after` field to the highest trimmed operation (in this case: 3) the storage server is instructed to mark $\boxed{2}$ as junk and insert the operation in the next available slot $\boxed{5}$.

RB-Log also sets the `insert_after` field to be the last sent operation when no trim has occurred, as a situation in which an operation is assigned a GSN lower than their last operation while safe is counter-intuitive (and would only occur with IM operations due to its implementation).

4.2.5 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 [18]. 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.

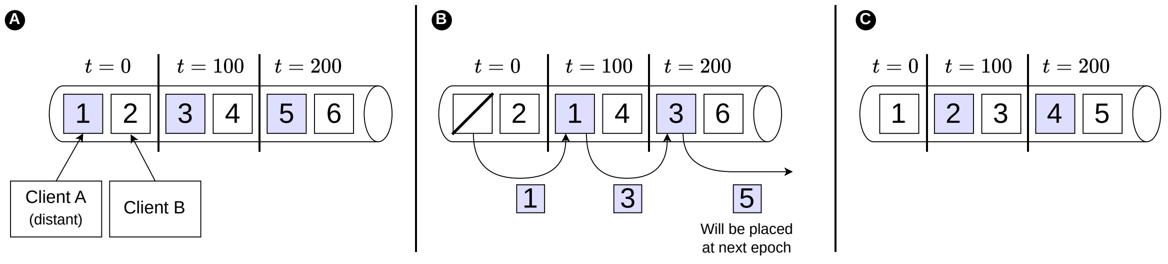

Figure 4.3: Example IM operation reorder.

4.2.5.A Latency Compensation and Delayed Serialization

As explained in Chapter 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 4.3 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 **A** we can see that while both operations $\boxed{1}$ and $\boxed{2}$ are produced at the same time. However, due to the network delay between regions, operation $\boxed{1}$ 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 **B**, resulting in the serialization illustrated in part **C**. 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.

Summary

This chapter presented RB-Log, a prototype implementation of the architecture introduced in Chapter 3. It described the baseline system, ZipLog, and how it was modified to support various operation types.

5 Evaluation

Contents

In this chapter, we present results from a series of experiments we performed to evaluate RB-Log. We will use the original ZipLog [7] implementation as a baseline.

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 Network Interface Card (NIC) and Ubuntu 22.04.4 LTS. The average Round Trip Time (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 [24], microsecond-level clock synchronization is now easily achievable across datacenters from cloud providers such as Google Cloud [25].

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 emulated topology is represented in Figure 5.1. The latency values were derived from the measured RTTs obtained in the experimental setup of Olissipo [3].

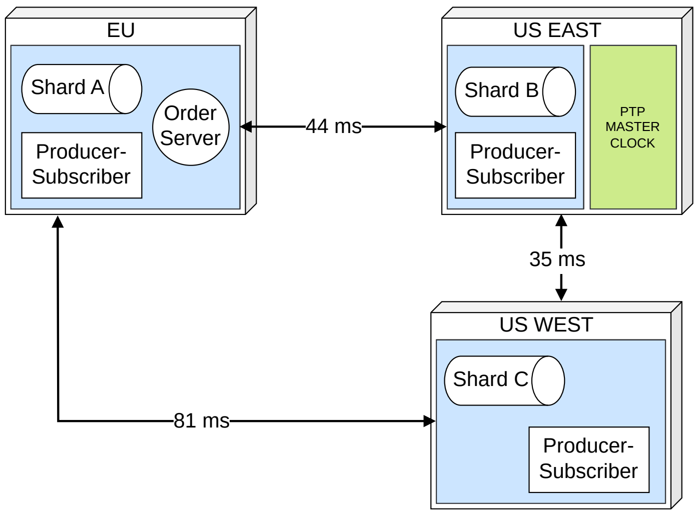

Figure 5.1: Experiment test setup.

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 (E2E) latency 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 Operations per second (OPS). 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. Figures 5.2 and 5.4 compare the median, 90th and 99th-percentile end-to-end latency and waiting time at the subscriber. Figures 5.3 and 5.5 show the median throughput and the median length of the fine-grained dependency section, respectively.

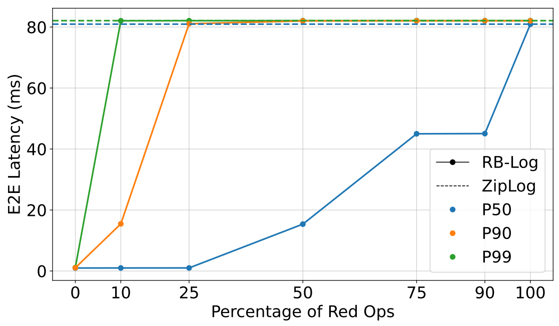

Figure 5.2: Median operation latency per subscriber (all regions combined).

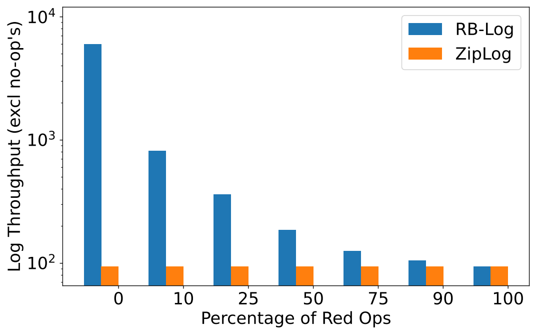

Figure 5.3: Median subscriber throughput (mean of all regions).

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.

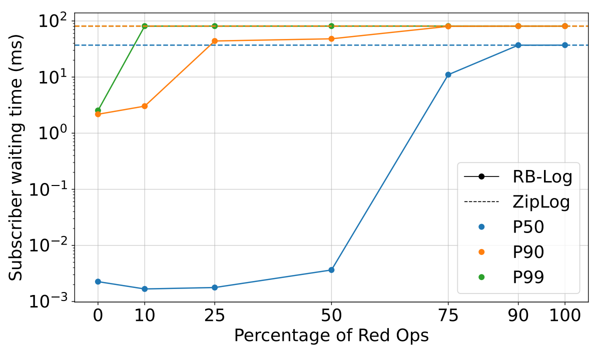

Figure 5.4: Operation waiting time at the subscriber (all regions combined).

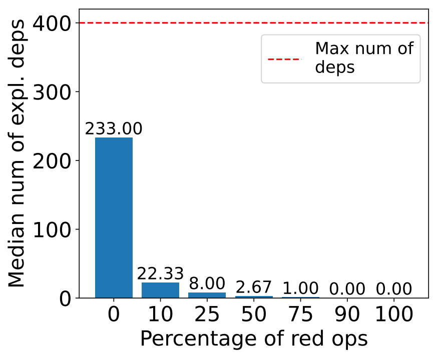

Figure 5.5: Number of blue fine-grained dependencies per red operation percentage.

The same tendency can be observed in the median waiting time where, in the median case, RB-Log is able to process operations at least three orders of magnitude faster than the baseline, thanks to the concurrent processing of blue operations.

Introducing Red operations significantly reduces both the number of explicit dependencies and subscriber throughput, by roughly an order of magnitude. At 0% Red operations, 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.

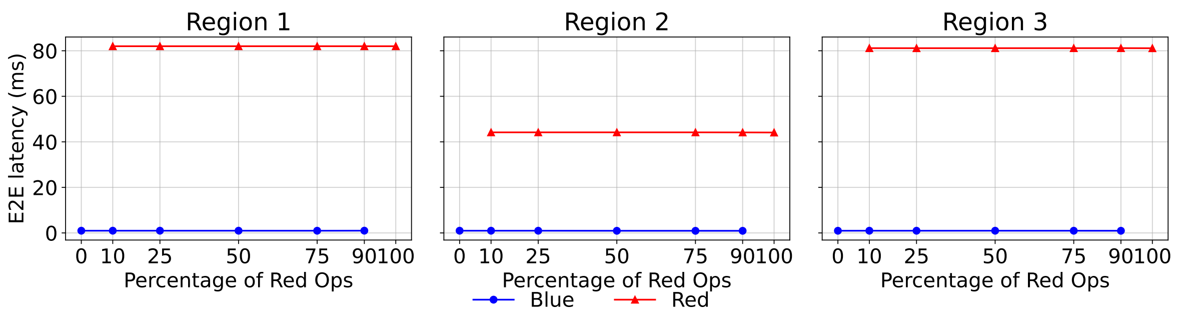

Figure 5.6: Median operation latency for Red/Blue per region and per operation type.

We further examine the impact of our emulated network delay in the system by plotting the median latency per color and region as shown in Figure 5.6. 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 $80ms$ for region A and C and $40ms$ 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 Chapter 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.

5.4.1 Impact of Latency Compensation

To evaluate the effectiveness of this technique, we use the same setup as in the previous section, but activate the IM stub at a rate of 1000 OPS, matching the Red/Blue stub. We then set the percentage of blue operations to 100%, causing the IM stub to send NO-OPs instead of real operations. This is advantageous in the experiment, as NO-OPs will not interfere with blue operations if the technique is effective. We set the maximum latency field to the highest observed network latency between each pair of regions, plus 1 ms to account for any delays. We then disable latency compensation. Figure 5.7 shows the median E2E latency for blue operations, with and without latency compensation.

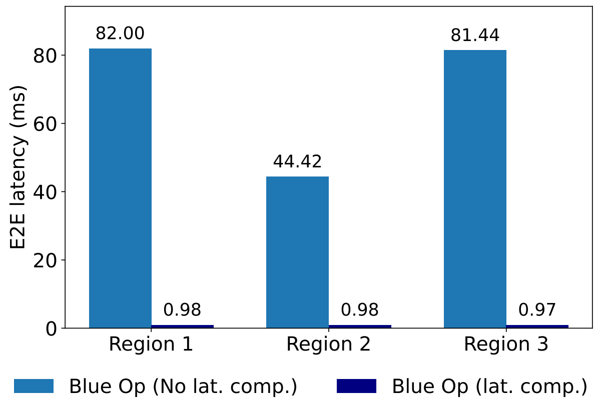

Figure 5.7: Median operation latency per region and with/without network compensation.

We can observe that the compensation mechanism allows the E2E latency of blue operations to remain low. This is possible because, as the IM NO-OPs reach the subscriber, they are marked as "processed", allowing the processing of the subsequent operations. With latency compensation, when a blue operation is produced the IM slots before it will already be filled with the NO-OP, hence it will be processed immediately. In the case of no compensation, IM operations will be produced and serialized along the blue operations, preventing them from being applied until they are received, thus the latency of blue operations corresponds to the maximum network delay. We thus conclude that the latency compensation mechanism prevents the system from halting in case of non-existing IM operations.

5.4.2 Impact of IM on Blue Operations

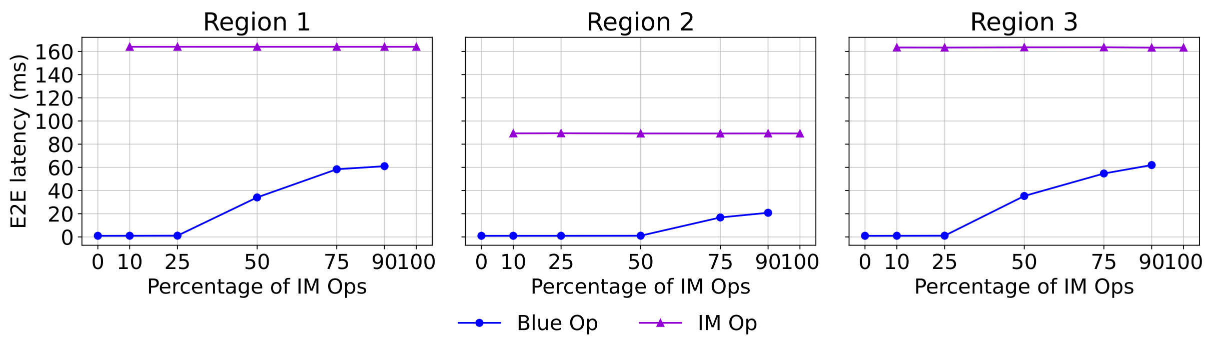

Figure 5.8: Median operation latency for IM/Blue per region and per operation type with compensation.

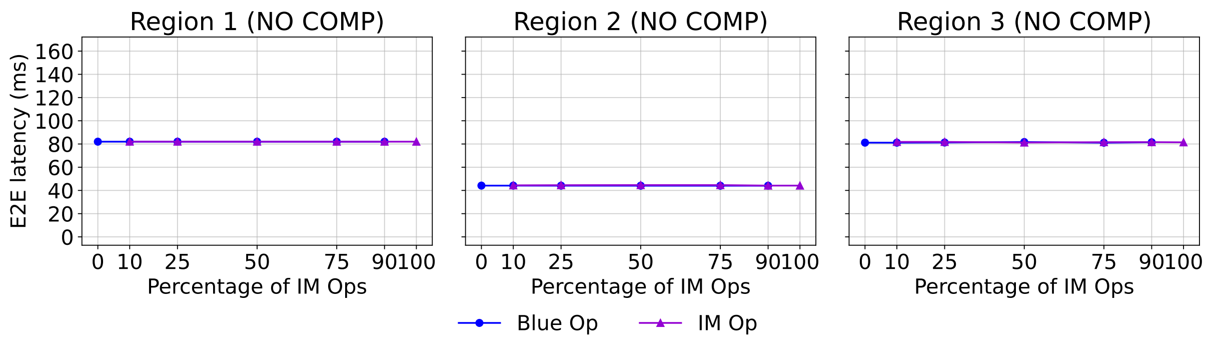

Figure 5.9: Median operation latency for IM/Blue per region and per operation type without compensation.

We now repeat the same experiment, but this time we vary the proportion of IM/Blue operations (following a uniform distribution) with and without latency compensation. Figure 5.8 and Figure 5.9 shows the median E2E latency per region and per operation type with latency compensation and without latency compensation, respectively.

From region 1, we observe that our mechanism 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 4.2.5.A, 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.

5.5 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 5.10 and Figure 5.11 presents the median E2E latency for all regions (averaged) and each jitter value. Additionally, Figure 5.12 shows, for each jitter value, the median latency per operation type. We also run each experiment without any network jitter, denoted “baseline”, which correspond, respectively, to the experiences for Section 5.3 and 5.4.2.

It can be seen that as the jitter value increases, so does the median latency, which affects primarily Red and Immediate 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 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.

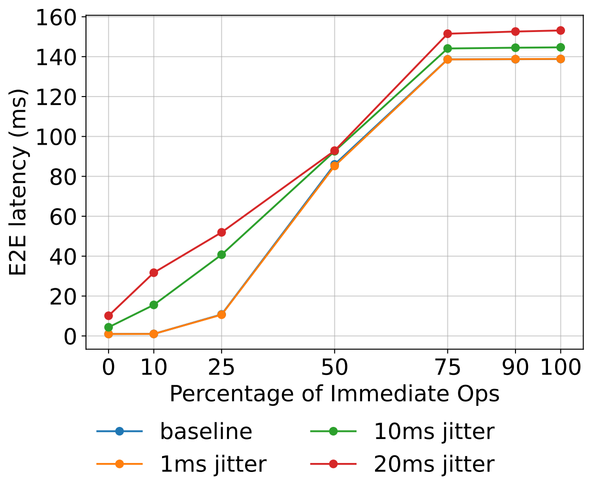

Figure 5.10: E2E Latency for Blue/IM operations with jitter.

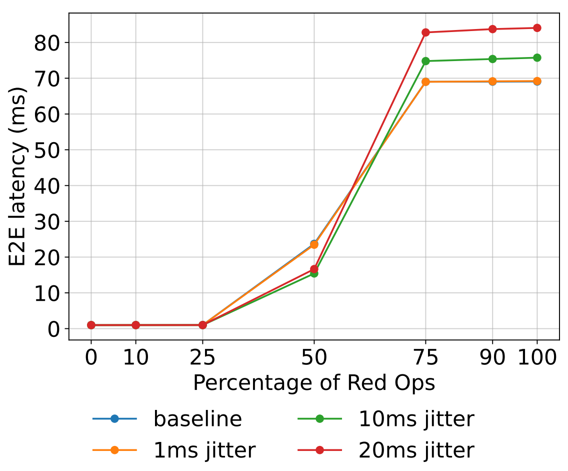

Figure 5.11: E2E Latency for Red/Blue operations with jitter.

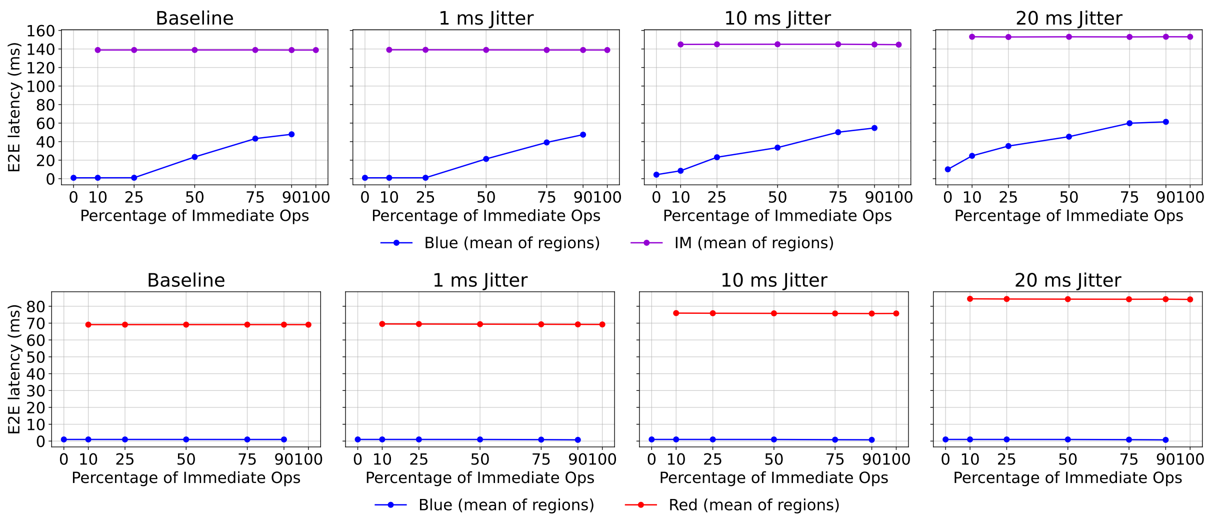

Figure 5.12: Median E2E latency for IM/Blue (top) and Red/Blue (bottom) operations for different jitter values.

Summary

This chapter presented the evaluation of RB-Log in a geo-distributed setting. The results show that RB-Log outperformed the baseline implementation, and its compensation mechanisms effectively reduced subscriber halting in the presence of IM operations.

6 Conclusion

In this dissertation, we reviewed the state of the art in systems supporting multiple consistency levels and shared-log architectures, discussing their main strengths and limitations. 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.

Bibliography

[1] “Apache kafka,” https://kafka.apache.org/, Accessed: 10/01/2025.

[2] “Logdevice,” https://logdevice.io/, Accessed: 10/01/2025.

[3] C. Li, N. Preguiça, and R. Rodrigues, “Fine-grained consistency for geo-replicated systems,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 359–372. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/li-cheng

[4] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, “Making Geo-Replicated systems fast as possible, consistent when necessary,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 265–278. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

[5] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan, “The FuzzyLog: A partially ordered shared log,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 357–372. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/lockerman

[6] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741972

[7] C. Ding, “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.

[8] “Jepsen consistency models,” https://jepsen.io/consistency/models, Accessed: 10/01/2025.

[9] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, 1994. [Online]. Available: https://doi.org/10.1109/PDIS.1994.331722

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual consistency," *Commun. ACM*, vol. 57, no. 5, p. 61–68, May 2014. [Online]. Available: https://doi.org/10.1145/2596624

[12] 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. [Online]. Available: https://doi.org/10.1145/78969.78972

[13] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 405–414.

[14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[15] "Dynamodb read consistency," https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html, Accessed: 10/01/2025.

[16] M. Bravo, A. Gotsman, B. de Régil, and H. Wei, "UniStore: A fault-tolerant marriage of causal and strong consistency," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 923–937. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/bravo

[17] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis, "Automating the choice of consistency levels in replicated systems," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 281–292. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, p. 360–391, Nov. 1992. [Online]. Available: https://doi.org/10.1145/138873.138877

[19] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "CORFU: A shared log design for flash clusters," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[20] R. V. Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-throughput-and-availability

[21] 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 *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 325–338. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/ding

[22] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.

[23] "Zeromq socket api," https://zeromq.org/socket-api/, Accessed: 6/10/2025.

[24] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "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*. USENIX Association, 2018.

[25] J. Geng, S. Mu, A. Sivaraman, and B. Prabhakar, "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.