Shared Logs With Support for Multi-Consistency

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

Francisco Carvalho — 99219*
franciscojcarvalho@tecnico.ulisboa.pt

Advisor: Luís Rodrigues

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. This report surveys the design of distributed shared logs and studies how they can be used efficiently by applications that support operations with different consistency requirements.

Keywords — Distributed Systems, Shared Logs, Consistency

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

Contents

1 Introduction

Shared logs are a key component of modern distributed systems with industry proven systems built on top of it like Apache Kafka [1] or Facebook's LogDevice [2]. The simplicity of a log abstraction where multiple clients append to a totally ordered sequence of operations makes them good choices for distributed applications that require strong consistency.

Despite this, strong consistency comes at the cost of performance, a critical requirement for large scale systems, leaving application developers to make a choice. In some applications, strong consistency is a non-negotiable requirement to ensure application correctness, while in others, the application semantics allows them to choose a relaxed consistency model, reducing coordination and improving performance.

There are, however, cases where applications only require some operations, but not all, to be strongly consistent. If a system 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 decide which operations are required to use strong consistency without significantly impacting the performance of the remaining ones.

This project is based on the observation that none of the existing state-of-the-art multi-consistency systems implement a total order via shared logs. We hypothesize that introducing weaker consistency levels over a log while keeping strong operations could make for a simpler architecture while yielding performance gains in log application throughput.

Thus, we would like to apply this strategy and bring causal consistency to a shared log, maintaining global order but allowing causal operations to be played by the applications as fast as causally possible.

The rest of this report is organized as follows: In sections 2 we present the goals and expected results of this work. In section 3 we present the necessary background information for the understating of the existing systems, presented in section 4. In section 5, we provide an overview of the proposed solution, along with its architecture and design decisions. Section 6 contains our planned evaluation methods. In section 7 we show the timeline for the execution of this work, concluding the report with a brief summary in section 8.

2 Goals

This work aims to increase the rate of operations applied by clients in a shared log system. That is,

> *Goals:* We aim to develop a shared log system capable of supporting both weak and strong consistency, allowing weak operations to be applied by clients faster and keep strong consistency an option for the subset of operation that requires it.

To accomplish this goal, we plan to extend an existing state-of-the-art shared log system with support for causal consistency.

The project is expected to produce the following results:

> *Expected Results:* This work aims to produce i) a novel multi-consistent shared log implementation; ii) An experimental evaluation of the performance with a focus on operation processing.

3 Background

In this section, we will present background information necessary to understand the next sections.

3.1 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.

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*, allows a client to request an entry, or range of entries to be discarded when 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.

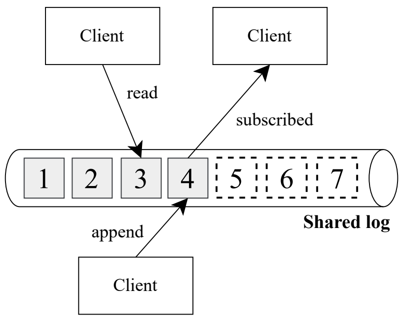


Figure 1: Shared log

Figure 1 illustrates a shared log systems, 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 shared logs provide a simple architecture for strong consistency, the write-once semantic for each log position also introduces the problem of contention, where multiple clients attempt to append an operation in the same log position (the tail of the log). Since only one operation can successfully be appended, the abort rate is high. This problem is addressed in the 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 appended

We will provide more details about these methods in the Related Work section.

Furthermore, as the log is just an abstraction, there are various ways of storing and replicating the entries according to the needs of the system. Most systems use a combination of data sharding and data replication. Sharding partitions the entry-space into various storage servers, utilizing a mapping for resolving log entries. Replication means making a full or partial copy of all the entries and placing it elsewhere, either to provide fault-tolerance and/or to geo-distribute the log and thus provide service to clients with lower round trips.

3.2 Consistency Models

As previously introduced, geo-replication allows the same data item to be placed in various regions so as to tame the time it takes to access it, among other advantages such as fault tolerance. However, as data is required to be distributed among different places and propagation is not instantaneous, different regions may have stale copies of the data which, if not accounted for, can result in application errors or erasure of data. It is then 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, in this section we will present some of the consistency models used by the systems in the related work section. More models can be found on the Jpsen website [7] (the consistency model descriptions presented below were partially adapted from this source).

Consistency models can be divided into 2 groups, those that apply to single operations and the models that govern the execution of transactions (which are groups of operations that are applied atomically). Informally, a consistency model can also be labelled as strong, if the model is restrictive and requires data to be consistent, or weak if it's more permissive, allowing replicas to observe stale copies of the data. One example of weak consistency is *Eventual Consistency*, used, for example in Dynamo [8], where it is possible for two replicas to temporarily diverge in state, provided that eventually they will converge to the same value. An example of strong consistency is *Linearizability*, defined below.

3.2.1 Single-object Consistency models

Read your writes ensures that, for the same process, every read that happen 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 cannot 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 the same process makes two sequential writes, then all processes must observe the writes in the submission order.

Causal Consistency guarantees the *happens-before* relation [9] (represented as $\rightarrow$). Under this model, causally-related operations should appear in the same causal order at every replica. That is, if a happened before b ($a \rightarrow b$) then b can only be seen after a. There are no guarantees of ordering for concurrent events, that is, events for which a relation of happens-before cannot be established.

Causal+ Consistency or "causal consistency with convergent conflict handling" [10] 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 is the strongest single-object consistency model. Under linearizability, all operations execute atomically and in a total order that is consistent with the real time, this means that all operations will be ordered, even concurrent operations. In practice, this model ensures the system behaves as a non-distributed, non-concurrent, system.

There are other consistency models that are relevant to our work, namely PoR and RedBlue consistency. These models will be presented in more detail, alongside their systems, in the Related work section.

3.2.2 Transactional Consistency Models

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

Transactional causal consistency introduced in Cure [11], extends the Causal+ consistency model to transactions. Under this model, all transactions execute their operations from the same causal consistent snapshot which includes all the operations in the issuing client's causal past. The transaction is executed atomically (either all writes are visible or none). As it derives from causal+, this model assumes that it is possible to merge together two concurrent transactions (specifically, using CRDTs [12]) so as to guarantee state convergence.

Serializability ensures that the concurrent execution of multiple transactions is equivalent to the execution of those transactions in some serial order. It requires a total order of execution for 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.

3.3 Multi-Consistency

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

As noted in Section 1, application developers traditionally had to choose between strong consistency and high throughput. Systems providing strong consistency cannot achieve the same performance as their weaker consistency counterparts. This is because strong consistency requires a total order, forcing sites to coordinate, whereas in eventual consistency systems, operations can be executed with little to no coordination. Some applications are able to tolerate the weak consistency with its drawbacks of stale copies and data reconciliation, thus enjoying the additional performance and scalability. Others cannot afford it, paying the performance penalty.

There are, however, cases where strong consistency is only required in a small subset of operations to ensure certain properties of the system always hold. Using the example from RedBlue [4], if a bank application cannot ever allow a negative balance, the withdrawal operation requires synchronization, as concurrent executions may violate such requirement. Despite this, other operations such as deposits are safe to be executed under weak consistency, as they will never lead to a negative balance.

Multi-consistency systems have been introduced to allow performance gains in systems where strong consistency is required in a subsection of operations, allowing the remaining operations to execute under faster weaker consistency. This idea of supporting strong operations has been partially implemented in some popular systems like Amazon DynamoDB [13] which, under certain conditions, supports strong consistent reads. Other systems like Gemini [4] have been designed to support multiple levels of consistency from the ground up, allowing developers to set the consistency type for each operation in the system. Other systems, like Unistore [14], 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 providing a good balance between consistency and performance, multiple consistency systems place the additional burden on application developers of determining which operations should execute in each of the consistency models. Some research has focused on finding ways to automate such task, with SIEVE [15] being one of such systems.

In Section 4 we will dive deeper on multi-consistency systems.

4 Related Work

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

4.1 Systems Supporting Multiple Levels of Consistency

Lazy Replication [16] describes a system for replicating state across multiple servers. Clients issue operations against a single replica which is then propagated to other replicas. The system supports three different types of operations. *Causal* operations execute under causal consistency while *forced* and *immediate* operations use strong consistency.

In *causal* operations, no coordination among replicas is required to accept an operation: any replica can accept a request and which is then propagated, in the background, to other replicas using a gossip procedure. Whenever a new operation is sent to a replica, it contains the issuing client seen operations as causal dependencies. These dependencies are represented in a vector clock sent with the operation. In this structure, every entry contains the number of the most recent operation observed that originated in a specific replica. 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, which stores the operations it has seen. The append process is then finished with the client adding the newly-appended operation to its vector clock. The operation will remain in the *log* until it is ready to be applied in the local state, i.e. when all its causal dependencies (captured in the vector clock) have been applied locally. The applied operation will then be propagated via the gossip protocol to the remaining replicas, where, similarly to client appends, they will be placed in their respective *log* until their dependencies are applied. Causal consistency thus allows the system to provide high-availability for operations that do not require strong semantics.

Forced operations require a total order between any other forced operation. This is implemented by extending the vector clocks with a new entry that is incremented with every new strong operation, thus totally ordering them. Every replica is then required to apply all operations with a lower component before executing a given forced operation. Furthermore, to avoid concurrent updates having the same number, only one replica at a time can be responsible for its increment. In this system, write access to the strong component is regulated using a Primary-Backup model with view changes. In every view there is an elected coordinator who gathers information from a majority of replicas, such that it can compute the highest sequence number assigned to a previously committed operation. This ensures that the new primary can safely assign sequence numbers from this point to new operations. It then proceeds to propose a value for the pending operation, applying it after a quorum of acknowledgments. The client is then informed and the commit information is propagated to the remaining nodes via gossip.

Immediate operations impose a total order on all operations (both *causal* and *immediate*). These operations will slow down the *causal* operations as they require a temporary halt of requests due to the total order between all operations. Immediate operations are also executed by the Primary-Backup view (assuming the view includes all replicas) and involve system-wide synchronization using a 3-phase protocol.

Gemini [4] is a fully 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 eventual 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:

- 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.

- Blue operations operate under causal consistency. Remote blue operations can be immediately applied as long as all their causal dependencies have already been applied. This means that different replicas may 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, we only can make an operation blue if their concurrent execution does not break any of the application's invariants.

Each site maintains a local causal serialization, which is a total order of all operations executed at that site. As such, locally appended operations will then causally depend on the last operations executed at that site.

Having a restriction such as globally commutativity ensures eventual global state convergence but, in turn, restricts the set of possible blue operations. This gives applications no other choice but to mark those operations as red. The authors mention this problem, further noting that whilst some operations may not be globally commutative, the effect of their application in the state might be. Taking the example from the paper, a calculation of due interest is not commutative as it depends on the possibly stale balance, but the value that resulted from the calculation (equivalent to a deposit) 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.

This way, once a new operation is sent to a replica, the generator for that operation is executed locally, which then invokes its corresponding shadow operation. Shadow operations are the only operations propagated to other replicas.

The concepts explained above are implemented in Gemini as follows: 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 seen all the causal dependencies for that operation. If it passes, it is then forwarded to a *data writer* that will apply it into the state.

The local concurrency coordinator relies on timestamps in the form of vector clocks to check if a shadow operation is admissible. Each vector clock has a value indicating the latest operation originated at the remote site that was replicated locally. There is also an additional strong entry that tracks the totally ordered red operations (as in Lazy Replication). 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 by a single token, passed around the replicas, giving them the ability to append their red operations and replicate them.

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

PoR is based on the RedBlue consistency model, addressing one of its performance bottlenecks when handling certain real-world use cases. Specifically, PoR departs from RedBlue's single total order of execution in Red shadow operations. RedBlue, whilst effective at providing strong and weak consistency, is not ideal for operations that only need to be totally ordered against a small subset of the red operations and who end up paying the performance penalty of waiting for the single order of execution. In turn, PoR addresses this case by dropping the Red/Blue operations types, instead, allowing each operation to list the operations it needs to synchronize with (its conflict operations). 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 the 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 gives more flexibility for application programmers in implement 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 ordering service is composed of multiple *agents* that order append requests from their local data center. Coordination between two conflicting operations can be achieved in multiple ways (Paxos, Distributed locking, ...) to ensure flexibility, Olisipo provides two protocols for coordination of each of the conflicts:

- **Symmetric** where both operations are required to coordinate. This is implemented with the counter service running Paxos. For each conflict pair, two counters are stored representing the number of operations appended for each type. Local agents then cross-check their counters with the global service to check whether they applied all prior operations.
- **Asymmetric** assumes one operation will be executed more often than the other. It allows the most frequent operation to execute by default without coordination. This is implemented using a distributed barrier system. Once the non-default operation is appended, processing of the conflicting operation is halted until the operation is received locally.

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

Whenever a new operation is sent to a replica, the generator operation is executed (as in Gemini). The result is then forwarded to this system for serialization and certification. Under the Gemini system, Olisipo replaces the concurrency coordinator (that enforces the RedBlue order).

Unistore [14] 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 (each strong operation is tagged with a set of operations which conflict with it): one of those operations will have to see the other and thus there is a total order for each conflicting transaction that overlaps in the read/write set.

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 data item is stored as a log of updates.

Causal transactions execute locally in the data center where the client is registered and are then forwarded to the remaining data centers in the background, as with Cure [11], which is the base for this system. Every transaction executes in a causally consistent snapshot which includes all transactions that the client has submitted. This snapshot will ensure that all replicas applying the operations will have those dependencies recorded prior to executing the operation. When a transaction is replicated at $f + 1$ data centers it is considered durable. This is because, by that point, at least one correct data center will have received it and can forward it further in case of a failure. A transaction can only be made visible to a client when all its causal dependencies have been applied. Dependencies are encoded as vector and version clocks (similar to Cure's [11] stabilization protocol). Replicas periodically exchange these vectors to determine which operations are uniform and whether any forwarding is needed.

Strongly consistent transactions are optimistically executed in the local data center and are then submitted to global certification. The certification service will then ensure that no conflicting operations are executed concurrently (in the same data item). If a dependency for that same transaction is missing, a data center cannot expose the results of the operation to clients. This poses a liveness problem as a data center may fail and never send the dependency. To mitigate this, all dependencies of a strong operation must be uniform prior to committing the transaction. This guarantees that every dependency will eventually be forwarded by at least one correct data center. Since data centers only expose remote operations that are uniform, the client snapshot may be in the past. Therefore, a strong operation may only wait for the causal operations executed locally to become uniform (which need to be exposed to clients immediately after being committed as to maintain the *read your writes* semantic).

4.2 Systems Based on Shared Logs

Corfu [17] is a shared log system focused on providing strong consistency and performance using flash-based clusters. It uses a single append-only log, sharded across clusters. Corfu uses a client-centric approach in which clusters only store the entries and clients are responsible for the append and replication of the data. The log is an abstraction implemented as a set of positions in various storage servers. As such, clients have to maintain a single mapping of log positions to the physical entries in the SSDs (and their respective replicas), called a projection. Clients can append data directly to the last available log position. This is, however, inefficient as if multiple clients attempt to write at the last position, all of them except for one will abort. This property, while guaranteeing total order across the log, comes at the cost of introducing contention. To minimize 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 [18].

While using a sequencer and reserved positions avoids the contention problem, it introduces a problem of possible holes in the log due to network partitions or client failure, 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 combat this, Corfu allows clients to mark entries as junk if they suspect that the client has crashed or to attempt to complete the replication.

In case of a shard failure, the next replica in the chain will take its place and a new one will be appointed as its backup (tail). For this to happen, the client's projection needs to reflect the new topology. The projection version is tracked with a Paxos epoch number that is passed in all messages sent by the clients. Thus, changing the projection implies starting a new epoch. The procedure is launched by a client who first attempts to seal, for the current epoch, the flash units whose ranges of positions are about to change to another server, ensuring that all the keys will be copied to the new server and avoiding race conditions with pending writes. When a storage server is sealed, it will reject any new messages, rendering the system almost unavailable during reconfiguration. After all the required shards are sealed, the client tries to write its new version of the projection to the upstream. In cases where the reconfiguration was started to add a new cluster to the system, no further action is required. If, however, there was a shard failure, the data stored in the backup will have to be copied to a new one. This will happen in the background and when finished, a new reconfiguration is initiated to add that new mapping of the failed entries to the new backup.

Scalog [19] is a shared log system which tries to mitigate some of the drawbacks of Corfu, namely the total halting of the system during reconfigurations, the sequencer as a bottleneck and the need to run custom hardware.

This is achieved by not using a centralized sequencer. Instead, clients append entries to the local shards, where they are stored in a local FIFO ordering. From this, operations are also forwarded to be replicated in every storage servers within the shard. Then, from time to time, all servers send their operations to a group of servers running Paxos (collectively called the *ordering layer*). These servers 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 for every server it matches the last appended (primary) and received (backup) entry. This is possible as replication is done in FIFO ordering. These scalars, called the *local cut*, are then aggregated by the ordering layer to capture the durable operations of the entire system, called the *global cut*. The sequence of *global cuts* is then used for establishing a total order. By subtracting two consecutive *global cuts* we get the newly durable operations at every shard, then the shard and server IDs are used to sort these new operations, establishing a total order.

Operations in Scalog execute with the linearizable consistency model. Reads can be performed at any server within the shard but before any value is returned, the storage server must first check whether the log sequence number supplied by the client is smaller than the last operation observed by the server. This ensures the local data within the server is sufficiently up-to-date to retrieve that value. 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. Reconfigurations are initiated by the *ordering layer* and allow for the continuity of reads and writes (the latter only in non-finalized shards). A server failure is detected as it stops sending *local cuts*. 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.

Dapple (FuzzyLog) [5] is a distributed shared log for multi-site 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 updates to the same logical shard are organized into *colors*, with operations originated at the same site also belonging to a same 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 elsewhere. Such loose synchronization enables updates across regions to the same color to be performed concurrently (with no conflicts, as the updates are appended to different chains). If there is only one color and one region, FuzzyLog behaves as strong consistent shared log.

A client appends data to the tail of its local chain in the color of the selected shard. The operation also contains links to the last node seen by the client in each remote chain for that color (its causal past). The last operation seen by a client (the *client view*) may not be the last operation that arrived at the site as they must synchronize their *view* and play the arrived remote changes (in an order that respects the causality). When the client syncs the view, it also has to ensure it has all dependencies for each operation before exposing them to the application. In Dapple, the servers may fetch unreceived operations (like a client) to get those dependencies.

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 [18]. Additionally, all client operations require a coarse-grained lease from the corresponding server. The authors used the FuzzyLog to implement a number of systems. For example, *LogMap* is a system using 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, also supporting 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+.

4.3 Comparison

In this section we present a brief comparison of the systems presented earlier. 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.

All these systems are built upon the earlier systems. In this case, LazyReplication laid the groundwork for RedBlue 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 by 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

Table 1: Comparison of properties in shared logs

Table 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, however, 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 the chain replication model. Furthermore, a failure requires a reconfiguration to change the projection, which will halt the entire system. In Scalog, data is replicated in 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. 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. Under Scalog, operations are appended and ordered locally in FIFO order until they are fully replicated. Then, once they are sent to the ordering layer, they will be totally ordered against the already ordered operations, and by the shard ID and server ID among themselves. Dapple uses a variation of Skeen's algorithm [20] to totally order operations locally and, in case of CAPmap, operations are forwarded through a single primary server in order for a total ordered to be established.

Multi-consistency systems

We will compare the multi-consistency systems across 4 categories, summarized in Table 2 and described in more detail below.

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); reading from replicas
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 default operation with barrier for the other	Paxos-like SMR for counter service
Unistore	PoR	Vector Clocks with a strong op counter	Global certification with 2PC+Paxos	Operation retransmission; remote operations only visible once uniform

Table 2: Comparison of properties in multiple consistency systems

Consistency levels: In terms of consistency models provided, all four systems allow applications to execute operations within strong and 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).

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 in the literal sense. As a coordination service for PoR consistency, it sits on top of a system like Gemini. The metadata it keeps in the form of logical clocks are used to determine a total order for conflicting operations, not covering the remaining causal past of such operations.

Strong operation implementation: 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 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 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 uses a retransmission mechanism to ensure shard failures do not pose a liveness problem.

5 A Log with Support for Causal Consistency

We now propose a set of mechanisms to implement a log that provides RedBlue consistency, supporting operations with different consistency levels. Namely, it supports strong operations, that need to be totally ordered, and weak operations, that only need to be causally ordered. The advantages of weak operations are twofold: first, weak operations may be processed even if the log is incomplete, as long as all its causal dependencies are met. Second, they may also be processed concurrently, improving processing parallelization. Our goal is to reduce the observed latency when consuming weak operations and improve processing throughput.

5.1 System Model

We consider a system composed of multiple processes that interact with each other via a shared log. Clients can be *producers*, that append operations to the log, or *subscribers*, that read the log (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 in 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.

We support two types of operations, namely *strong* and *weak* operations (equivalent to the *Red* and *Blue* operations of RedBlue consistency), as described below.

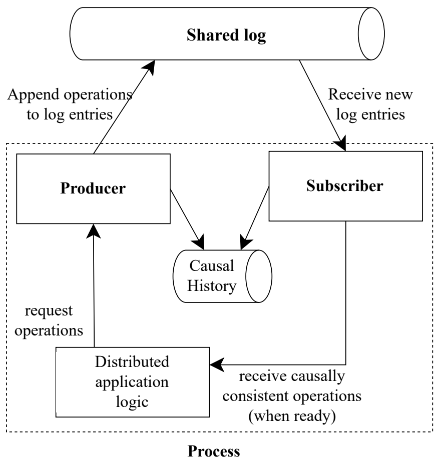
- **Strong Operations:** Strong operations need to be applied by all subscribers in the same total order. In our system, this total order will be the order defined by the underlying shared log. Strong operations may only be applied once all previously ordered strong operations have been applied and all causal dependencies have been met. To simplify its implementation, a strong operation may only be applied once all operations that precede it in the log have been made visible, so as to ensure no strong operation is missed.
- **Causal Operations:** Causal operations execute under the causal consistency model, which allows them to be applied immediately as long as all operations in their causal past have been applied. This allows a causal operation to be applied even if some of the previous entries in the log have not been made visible yet. Also, because different operations may become visible to different clients at different points in time, clients may apply causal operations in different orders. We assume that causal operations are "globally commutative" [4], such that all clients converge to the same state. Therefore, while causal consistency improves performance by allowing out-of-order execution, global state convergence ensures we can do so safely (by preventing permanent state divergence).

5.2 General Architecture

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

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

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



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 2: General Architecture

The condition that makes an entry *ready* depends on the type of operation. If the operation is tagged as strongly consistent, it becomes ready once all causal dependencies and all strong operations ordered before it have been added to the *applied entries* set. If the operation is tagged as weakly consistent, it only requires its causal past to have been added to the *applied entries* set.

Because the log is geo-distributed, entries may not become visible in the order defined by the log. As discussed in Section 4, log entries written by local clients may become visible immediately to subscribers in the same region, before log entries written by remote clients become visible (even if these entries are totally ordered before the local operation). This explains why an operation that is already visible may need to wait before being applied.

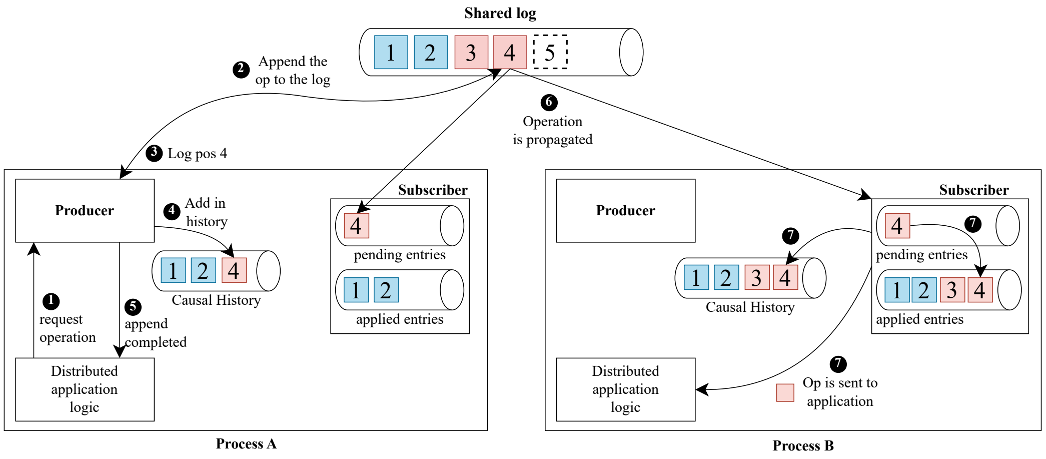


Figure 3: Operation with two replicas in different locations

Figure 3 illustrates the operation of the system with two processes that are both subscribers and producers. Weakly consistent operations are represented in blue and strongly consistent operations in red. The number in each operation corresponds to their position in the shared log and the applied entries list tracks the operations sent 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 a log position that is promptly appended to the causal history of the process, concluding the append (❸,❹,❺). The new entry will then 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.

5.3 Enforcing Causality

Whilst having causal and strong operations being processed allows for faster throughput compared to only strong operations, causal consistency requires finer-grained dependency management, which may require processing, possibly reducing the causal processing advantages if not accounted for. In this section, we will present 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 the highest log position observed by the client (either applied or appended by it). The newly-appended operation would then causally depend on all operations with a log position lower or equal than this specified position. Compared to using vector clocks, this method would make metadata as small as a single scalar and without the need for monotonic clocks. It has, however, the drawback of possibly making new operations dependent on ones not seen by the client (fake dependencies). This is because, as presented earlier, causal operation can be played out of log order. If a client has "holes" in its log, those missing operations would still be causal dependencies of any new operation due to the existence of an applied operation with a higher log position.

Vector clocks

On the other hand, 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. They are stored by each client and reflect the operations appended and applied by that specific client. Vector clocks allow replicas to encode which operations have been locally 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, 14, 16], they would require maintaining a view of existing processes in the system, incurring heavy costs on changes of clients memberships. We aim at a solution agnostic to the clients/processes view.

Explicit dependency tracking with top-most dependencies

As previously mentioned, keeping explicit dependencies is very expensive. We can opt for using explicit log positions as dependencies and try to keep them as small as possible. This can be done by removing redundant dependencies that are implicitly present as dependencies for other dependencies. By applying this, each operation will compute 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 should reduce the growing number of dependencies, but it cannot remove them entirely. (This optimization is formally defined in COPS [10]).

Hybrid dependency tracking

While the aforementioned optimization further 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

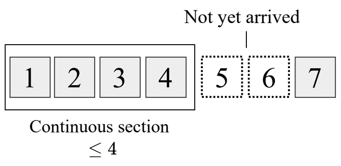


Figure 4: Example of a local log with a continuous section and two late operations

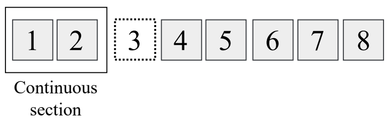


Figure 5: Example of how a late arrival of a single operation may make the explicit section much bigger

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 will be reserved for the operations that were executed in incomplete sections (to which not all operations with lower ordering have arrived). This approach allows the system to cut the number of explicit dependencies while still expressing fine-grained dependencies. It also keeps the client's view of the system consistent, as all operations on the continuous section have already been observed and are thus causal dependencies.

Figure 4 illustrates this technique. Consider the local log at a given replica. Greyed out boxes represent applied operations at the replica, while dotted ones are operations yet to arrive. Numbers show the total order provided by the global shared log. The causal dependencies for an operation to be appended in this current state would be represented as `{4,[7]}`, where 4 denotes the continuous section, which includes all elements lower than or equal to $\boxed{4}$, and the array denotes the explicit elements to be included. If operation $\boxed{5}$ were present, the dependency would have been represented as `{5,[7]}`.

One drawback of this implementation is the reliance on the ability of the client to fill all the holes in their log. This means that delays in a single operation may increase the number of explicit dependencies, defeating its purpose. Take Figure 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.

We address this by introducing a ceiling to the number of explicit operations in the timestamp. If the configured threshold - determined by a variable λ - is passed, 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 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.

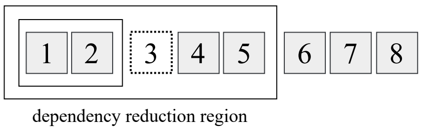


Figure 6: Example of applying a dependency ceiling

The optimal length for this upper bound is workload dependent, and the value can either be static or change dynamically according to collected metrics. During the course of the thesis, we will evaluate the impact of these various modes on performance.

5.4 Optimizations for Strong Operations

Some shared logs, such as vCorfu [21], have support for multiple streams. Our approach could leverage such support to optimize strongly consistent operations. This could be achieved by configuring the underlying log to support two different streams, one for each type of operation. In this case, we could extend the system as follows: when a strong operation is added to the global shared log, information regarding the strongly consistent stream could be used to extract the index of the previous strong operation in the global log. This index would be automatically added to the set of dependencies of the operation being added to the log. In other words, every strong operation would explicitly depend on the previous strong operation. Using this information, strongly consistent operations could just wait for their dependencies (instead of waiting for *all* previous entries to be filled). This may allow strongly consistent operations to be applied sooner.

We plan to evaluate whether these optimizations will impact performance in a significant way.

6 Evaluation

Our work can be viewed as a set of extensions deployed on top of a shared log system. Therefore, we plan to compare the performance of a system with our proposed extensions to that of the underlying log without the extensions.

Specifically, we aim to improve the local log processing throughput. Thus, we will evaluate how each proposed extension affects this metric, together and in isolation. Other metrics will be considered, such as the size of the dependencies and number of appends per second.

We plan to center our testing efforts around these key-points:

- **Strong/Weak Operation Performance:** We would like to understand how changing the number of strong operations affects the overall latency and throughput, testing the system with different ratios of strong operations (for instance, 100%, 30% and 0% of, values that have been used to test Gemini).
- **Dependency Resolution Time:** We will measure the complexity of processing the dependencies of operations as well as the impact of increasing the number of explicit dependencies. To accomplish this, we plan to disable the maximum length for the explicit section while introducing a propagation delay in one of the log entries. This will make all operations ordered after the delayed operation to be placed in the explicit section of the metadata. We will then introduce our dependency maximum length value and determine how much it impacts the system latency and throughput.
- **Strong Operation Optimizations:** We will measure how much impact the optimizations described in Section 5.4 will have on the overall throughput.
- **Overall Performance:** Finally, as stated at the beginning of this section, we plan to measure how our optimizations improved the log application throughput.

7 Work Schedule

A Gantt chart is shown in Figure 7.

8 Conclusion

Multi consistency systems emerge as a strategy to balance consistency and performance, offering strong consistency when necessary while maintaining higher throughput with eventual consistency. In this report, we reviewed the state-of-the-art in systems supporting multiple levels of consistency and systems using shared logs, discussing their main strengths and weaknesses. Additionally, we have presented a

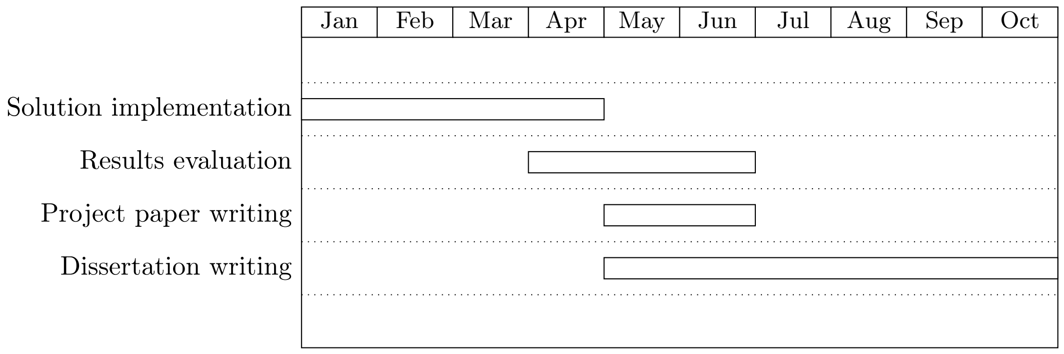


Figure 7: Planned Schedule

solution for extending an existing shared log with multi-consistency support. Finally, we presented our proposed evaluation methods as well as our work schedule.

Acknowledgments We are grateful to Rafael Soares for all the feedback provided during the writing of this report.

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] "Jepsen consistency models," https://jepsen.io/consistency/models, Accessed: 10/01/2025.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205–220, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[9] 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

[10] 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

[11] 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.

[12] 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.

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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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.

[21] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vCorfu: A Cloud-Scale object store on a shared log," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 35–49. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael