Design of Geo-Distributed Shared Logs to Support Partially-Replicated Transactional Systems

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

Inês Cardeira — ist198941*
ines.cardeira@tecnico.ulisboa.pt

Advisor: Luís Rodrigues

Abstract We are interested in the design of strongly consistent partially replicated transactional systems. In these systems data is split among multiple shards and different replicas store different shards. Shards accessed in the context of distributed transactions can read and write objects from one or more shards. We consider systems that enforce serializability and, therefore, require all transactions to be totally ordered. This can be achieved by submitting transactions via a totally ordered shared log. In this report, we survey existing techniques to order transactions in partially replicated systems, including systems that use shared logs for that purpose. We aim at finding techniques that can support some amount of concurrency when ordering transactions that access different shards.

Keywords — Distributed Transactions, Partial Replication, Shared Logs

*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

Many modern data stores are both geo-distributed and partially-replicated. These systems include multiple nodes (typically hosted in cloud data centers) that reside in different geo-locations, often separated by large network latencies. Data is split among multiple shards and different nodes replicate one or more shards. In many cases, it is required to support strongly-consistent transactional access to data, namely, to ensure serializability or snapshot-isolation. This requires transactions to be totally ordered.

In this report, we survey existing techniques to totally order transactions in partially-replicated systems. In recent years, many transactional systems have been built using a distributed shared log as a building block. Therefore, in our study, we will address the use of totally ordered logs to serialize transactions and compare these solutions to other ordering strategies that do not rely on shared logs.

1.1 Work Objectives

We aim at designing novel techniques to totally order transactions on geo-distributed partially-replicated systems. For this purpose, we will begin by analyzing the trade-offs involved when deploying different ordering algorithms in this setting. Based on the insights obtained by surveying the state-of-the-art, we will propose a novel architecture to order distributed transactions.

1.2 Expected Contributions

We expect to be able to propose a novel algorithm to totally order transactions in geo-distributed partially-replicated systems. Our algorithm will aim at reducing the impact that “global” transactions (i.e., transactions that access multiple shards in multiple locations) may have on the execution of “local” transactions (i.e., transactions that access a single shard, replicated in a small number of locations close to each other).

1.3 Expected Results

Our work will produce:

- A prototype of a geo-distributed partially-replicated transactional key-value store based on our new algorithm.
- An experimental evaluation of the performance of the resulting system against other state-of-the-art approaches.

2 Background

Understanding distributed systems requires familiarity with some key concepts that establish their design and operation. This section provides an overview of foundational concepts, including shared logs, log ordering techniques, transactions, consistency models, and metadata management. Each of these topics plays a critical role in maintaining the reliability, performance, and consistency of distributed systems.

2.1 Shared Log

A shared log is a distributed data structure that maintains a history of records, typically in a fault-tolerant manner. For this purpose, each record is stored in multiple storage nodes. In addition, log entries are immutable, and the only way to update the log is by adding new entries. This allows applications that feed from the log to keep a mutually consistent state. For instance, a log can be used to store commands to be executed by a state-machine: in this case, clients add commands to the log and state-machine replication can be trivially achieved by having different server replicas read the log and process commands in the order specified by the log. Shared logs have been used to support many distributed applications, including cloud service providers and other large-scale architectures, offering fault tolerance, state replication, and simplified system reconfiguration.

2.2 Log Ordering Techniques

Given that a log is replicated for fault-tolerance, replicas need to agree on the order by which entries are added to the log. This requires the execution of some form of consensus protocol with a dual purpose: replicas must agree on the set of entries that are added to the log and on the sequence number assigned to each entry. Different techniques have been proposed to assign sequence numbers to replicas, including:

Sequencers: This technique uses a centralized component, named a *sequencer*, that is responsible for ordering the operations of the system. Examples of systems that use sequencers are CORFU [1], vCorfu [2], and SLOG [3]. Sequencers avoid the costs of coordination that may occur when different replicas concurrently attempt to assign the same sequence number to different entries but may become a bottleneck in the system. Additionally, in geo-distributed settings, using a centralized sequencer may add a significant latency to the ordering procedure, since remote locations may observe large network delays when contacting the sequencer.

Stabilization: In this technique, a single replica assigns to each entry a local sequence number. Different replicas can assign local sequence numbers to different entries concurrently and without coordination. Then, local sequencer numbers are exchanged among replicas and deterministically merged to establish a global total order. To perform the merge, a replica must have received local sequence numbers up to a given point from *all* other replicas. Thus, if a replica has not locally assigned sequence numbers, it needs to notify all other replicas by sending special *null* message such that deterministic merge can be applied. When using stabilization, there is a trade-off between the overhead caused by null messages and the latency of deterministic merge.

Tree-Based Aggregation: This technique combines features of the stabilization and sequencer approaches. Similarly to stabilization based systems, different replicas can assign local sequence numbers to different entries concurrently and without coordination. These local sequence numbers are exchanged using a tree of nodes that act as message forwarders and aggregators, as they perform a deterministic merge of multiple sources before propagating a merged stream of sequence numbers. When the root node deterministically merges the streams coming from its children, it established the final total order to be used across all replicas. Some examples of systems that use such technique are Saturn [4] and ENGAGE [5].

2.3 Transactions

A transaction is a set of operations that are combined and executed as one cohesive atomic unit. Systems that support transactions often guarantee a set of properties, known as the ACID properties, namely Atomicity, Consistency, Isolation and Durability. In a distributed system, the execution of a transaction may involve multiple nodes that need to coordinate to ensure these properties. Two-phase or three-phase commit protocols are coordination protocols that can be used to ensure that all participants in a transaction agree on the outcome of the transaction, i.e., if the transaction commits or aborts, such that atomicity is preserved. Isolation is provided by the execution of some concurrency control mechanism. Many concurrency control protocols require transactions to be serialized and, therefore, require a total order to be defined. In this context, distributed shared logs can be a useful tool to serialize transactions and agree on their outcome.

2.4 Consistency Models

A consistency model defines what are the valid outcomes of the concurrent execution of multiple clients. Consistency models have been defined for scenarios where clients submit isolated operations and for scenarios where clients submit transactions. Various consistency models manage the balance between performance and availability, each with its own trade-offs. Informally, a consistency model is denoted to be "strong" if the outcome of the concurrent execution of multiple transactions is equivalent to the execution of those transactions in isolation, in serial order, in a single replica. As it will be discussed next, strong consistency requires operations to be totally ordered, and total order protocols may block in the presence of network partitions. This fact has been captured by the CAP theorem [6], which says that no distributed system can achieve at the same time (Strong) Consistency, Availability and Partition Tolerance.

Many consistency models ensure that operations are executed in an order that respects causality. Two operations are causally related if one operation influences or depends on the other. This relationship is captured by Lamport's "happened-before relation" [7]. For example, a write operation and a subsequent read of the same data are considered causally related.

In the following, we refer some key Non-Transactional and Transactional consistency models.

2.4.1 Non-Transactional Guarantees

Session Guarantees: [8] Set of consistency properties defined for weakly replicated systems. A session establishes a relationship between past operations performed by the client and the constraints applied to their future reads and writes, with the specific behavior depending on the set of guarantees the system provides within the session. Four guarantees have been defined, namely: *Read-Your-Writes*, that ensures that a client observes the effects of their own writes; *Monotonic Reads*, which guarantees that once a client reads a value for a given object, they will not observe an older version of that same object in future read operations; *Monotonic Writes*, that guarantees that all updates are applied in the order they have been submitted; and *Writes-Follow-Reads*, that ensures that an update is applied after all updates that have been previously observed by the client.

Causal Consistency: Based on Lamport's "happened-before relation" [7], it dictates that all replicas observe updates in an order that is consistent with causality, while allowing concurrent events to be observed in different orders.

Causal+ Consistency: [9] It extends causal consistency by also ensuring strong convergence. Conflicts in this context arise when different replicas concurrently update the same object, creating multiple possible versions. Causal+ consistency ensures that all replicas independently and deterministically resolve conflicts in the same way, reaching the same state.

2.4.2 Transactional Guarantees

Transactional Causal Consistency (TCC): It extends the concept of causal consistency by incorporating transactional semantics and ensuring atomic visibility of transactions. It guarantees that transactions operate on causally consistent snapshots, where all causal dependencies of the accessed objects are respected. Furthermore, TCC establishes causal relations between transactions, determining their order based on the causal dependencies of the objects they access.

Serializability: It ensures that the outcome of executing transactions in a distributed system is equivalent to an execution where all transactions are ordered sequentially, not necessarily matching the chronological order of transactions. In replicated systems, this property is also defined as **One-Copy Serializability**, ensuring that all replicas agree on the same global transaction order, producing results equivalent to executing all transactions sequentially on a single logical copy of the data.

Strict Serializability: It guarantees that the order of transactions across all the replicas of the system, reflecting their real-time order of execution. For this reason, since it imposes stricter constraints, it might lead to a worse throughput.

2.5 Metadata to Track Causal Dependencies

In distributed systems, metadata plays a crucial role in maintaining consistency by tracking dependencies between operations. It ensures that updates are applied in the correct order across replicas, respecting causal relationships. The choice of metadata design introduces trade-offs between storage overhead, computational complexity, and the system's ability to minimize false dependencies [4]. False dependencies occur when independent operations are incorrectly treated as dependent, causing unnecessary delays in making updates visible.

Below, we explore two common types of metadata used in distributed systems, Lamport's clocks and vector clocks:

Lamport clocks: Lamport clocks [7] are scalars that are used to keep track of causality. The rules to increment Lamport clocks ensure that, if an event e_1 is in the causal past of another event e_2, then the Lamport clock of e_1 is guaranteed to be smaller than the Lamport clock of e_2. However, it is also possible that the clock of e_2 is larger than the clock of e_1 without e_1 being in the causal past of e_2. This is called a false dependency, that results from the fact that it is impossible to determine if two events are concurrent from scalar clocks alone. False dependencies may introduce unnecessary latencies when making updates visible.

Vector Clocks: A form of logical clocks that uses multiple entries, usually one for each process or replica in the system, with each one of them maintaining its own vector clock. By precisely capturing dependencies, they minimize false dependencies, enabling operations to proceed independently when no causal relationship exists, which leads to a lower visibility latency as updates can be applied without unnecessary serialization. However, this approach comes with its own disadvantages. For example, the fact that it can lead to higher storage and computation requirements, making it ill suited for systems that prioritize high throughput.

3 Related Work

In this section, we will explore prior approaches to transaction processing and data management in distributed systems by focusing on three key areas: shared log architectures, locality in distributed logs, and tree-based architectures. These systems address the challenges of consistency, scalability and efficiency in distributed environments. Each subsection highlights the design principles, trade-offs, and innovations.

3.1 Shared (Totally Ordered) Logs

We start by presenting totally ordered shared log systems, where every operation in the log is assigned a unique position in a global sequence, imposing a strict, deterministic order of all operations.

3.1.1 CORFU

CORFU [1] is a totally ordered distributed shared log implementation that uses a storage system composed of multiple SSDs, where each SSD is responsible for storing a different portion of the log. Therefore, different clients can concurrently read or write in different portions of the log without interfering with each other. Furthermore, every portion of the log is replicated in multiple SSDs that are organized in a chain. Which SSDs keep each portion of the log is defined by a centralized component in what is called a *projection*. This centralized component ensures that all participants use the same projection. Individual SSDs are augmented with a hardware mechanism that ensures that each entry of the log can only be written once. Therefore, if two clients concurrently attempt to write on the same entry, only one of the clients will succeed.

Writing in the log consists of two phases: first the client must find the most recent unused entry in the log (i.e., the head of the log) and then the client must write in all replicas of that entry. To find the log head, the client may simply parse the log until it finds the first unused entry. Unfortunately, if multiple clients do this concurrently, they will all compete for the same entry, only one of them will succeed, and the rest must repeat the process. To avoid having multiple clients competing for the same entry, CORFU uses a centralized sequencer that is responsible for assigning log entries to clients. Therefore, a client first contacts the sequencer and then performs the append operation. Updates are performed by writing in all replicas, in sequence, starting at the head of the chain. The update is complete once the tail replica is written. If a client fails after being assigned a log entry, but before it writes to all replicas, a special procedure needs to be executed to "fill" unused or partially written entries.

By leveraging the client-driven chain replication protocol and a centralized sequencer for log position assignment, CORFU achieves high throughput, enabling clients to append at the maximum rate allowed by the sequencer's position assignment speed.

3.1.2 vCorfu

vCorfu [2] is a cloud-scale object store built over a shared log abstraction, designed to keep the advantages of systems like CORFU, such as strong consistency, while addressing their inherent limitations, including restricted read performance, through the use of materialized streams. A common issue in traditional shared log systems is the playback bottleneck, where clients must replay all updates in the log - including those irrelevant to their requests - to retrieve specific data. This often forces clients to contact multiple nodes, increasing latency. To solve these issues, vCorfu leverages materialized streams, which organize updates into separate streams, each corresponding to a specific object. These streams act as logical partitions of the global log, containing only updates relevant to their associated object. This allows clients to directly access relevant updates without traversing the global log, significantly simplifying read operations.

Similarly to CORFU, vCorfu is composed of a sequencer and a layout layer that maps log and stream addresses to the corresponding replicas, providing clients with a global view of the system. The sequencer extends CORFU's functionality by tracking the tails of both the global log and individual streams, maintaining entry counts for each stream, ensuring a total order of updates. When a client appends data, the sequencer assigns global and stream-specific addresses. Using these addresses, the client writes updates directly to the respective log and stream replicas. Once the updates have been successfully written, a commit message is broadcast to all accessed replicas, which guarantees that stream replicas are free of gaps, enabling efficient data access. This commit process is specific to the log operations, ensuring durability and consistency at the log level.

vCorfu also supports optimistic transaction execution on clients' in-memory views. These views allow the client to cache the latest state of the objects it interacts with. During a transaction, the client tracks the versions of accessed objects - its read set - and the modifications made to objects - its write set. The sequencer, acting as a lightweight transaction manager, validates transactions by ensuring that the read set remains unchanged. If validated, the sequencer issues new tokens, allowing the client to commit updates atomically; otherwise, the transaction is rejected. This mechanism ensures snapshot isolation and prevents unnecessary writes to the log, optimizing system performance.

The introduction of materialized streams enables vCorfu to achieve better read scalability compared to traditional shared log systems, where clients must replay the global log for updates. By allowing direct reads from stream replicas, clients can retrieve individual updates or entire streams with minimal latency. However, while the additional commit messages improve read efficiency, they introduce some overhead to write operations, which can reduce throughput in write-heavy scenarios. Nonetheless, this trade-off is justified by the significant gains in read performance and scalability, particularly for workloads with frequent reads or large data sets.

3.1.3 Scalog

Scalog [10], just like CORFU, is a totally ordered shared log abstraction. While CORFU requires clients to obtain a global sequence number directly from a centralized sequencer, Scalog moves this responsibility onto the shards themselves, allowing for further batching opportunities, reducing the load onto the centralized sequencer.

Clients in Scalog can write directly to the storage server of their choice without knowing the record's global sequence number beforehand. Upon receiving a record, the server stores it locally and begins the replication process within its shard. Scalog's architecture is structured in two primary layers:

- Data Layer: Comprises a collection of shards, with each shard storing and replicating records received from clients. Within each shard, each storage server acts as both a Primary for the records it receives directly from clients and a Backup for records in the log segments of its peers in the same shard.
- Ordering Layer: Responsible for merging the locally ordered logs from each shard into a single global total order. To achieve this, the layer uses a hierarchical structure of aggregator nodes. Each leaf aggregator collects and merges records from a subset of storage servers. Higher-level aggregators then merge the outputs from lower levels, reducing the workload at any single node and enabling scalability to the system across multiple layers of the tree.

The global total order is determined using a mechanism called the **Global Cut**, a vector representing the durable record count for each shard. Periodically, storage servers issue information to the ordering layer regarding the number of records stored. The ordering layer computes the minimum count reported among all servers in each shard, forming the **Global Cut**, which is then broadcast to all storage servers. Each server applies a deterministic algorithm to assign sequence numbers to records included in the cut, ensuring all shards independently derive the same global order without additional coordination. This guarantees that every record has a unique global position even if written to different shards.

This persistence-first approach allows Scalog to seamlessly reconfigure without downtime or data loss. By decoupling data persistence from global ordering, Scalog avoids the bottlenecks associated with centralized sequencers and mitigates the downtime availability during reconfiguration. The hierarchical ordering layer further enhances scalability by distributing the computational load of ordering across multiple aggregator nodes. Clients are notified of their record's global sequence number once it is fully replicated and ordered, ensuring durability and consistency. However, while this architecture achieves high scalability and throughput, traversing the aggregator tree and its reliance on periodic cuts from the ordering layer introduces a large latency overhead.

3.2 Locality in Distributed Logs

Although previous systems provide strong consistency, they can be expensive and inefficient in geo-replicated systems, since a total order often requires significant coordination between distant regions, leading to high latency.

This section discusses systems that do not impose a global total order. Rather, they use locality to partition data across regions, making them well-suited for geo-replicated environments. However, it also introduces trade-offs related to consistency and conflict resolution.

3.2.1 FuzzyLog

FuzzyLog [11] is a shared log abstraction that departs from traditional total ordering by employing a partial order of updates. This design choice enables scalable geo-distributed applications while offering flexible consistency guarantees, such as causal+ consistency. The system does not offer strict serializability, instead focusing on serializability as a trade-off for improved scalability and availability.

At the core of its architecture is a Directed Acyclic Graph (DAG), where nodes represent updates and edges encode the happens-before relationships. The DAG is divided into colors, each corresponding to an independent shard of application data. Within each color, updates are organized into chains that are fully replicated across regions. Each region maintains its own local chain and lazily synchronizes with the chains of other regions, ensuring that operations can proceed independently with minimal cross-region coordination.

Clients interact with their local DAG replicas to either append updates or synchronize state. For single-color operations, a client appends a new update by specifying its content and the target color. This update is appended to the tail of the local chain for that color, with cross links to the last nodes observed by the client in other chains. These cross-links ensure causal dependencies are preserved, enabling serializable isolation across shards. Synchronization for a specific color involves fetching updates from other regions in reverse topological order, ensuring that the causal relationships encoded in the DAG are respected.

For multi-color transactions, FuzzyLog employs a coordination protocol that uses Skeen's algorithm [12]. This protocol ensures that updates spanning multiple shards are applied atomically and in a globally consistent order. The process involves two steps: first, each participating server proposes a timestamp for the transaction, based on its local logical clock and a unique nonce. The initiating client collects these proposals and selects the maximum timestamp as the global order for the transaction. In the second step, the client disseminates this global timestamp back to the servers, which finalizes the transaction's placement in the DAG.

FuzzyLog delegates the responsibility for managing this coordination to clients, rather than a centralized sequencer, which aligns with its decentralized and scalable design. However, this approach introduces additional challenges, particularly for fault tolerance, as client failures can leave transactions in an incomplete state. To address this, FuzzyLog implements mechanisms such as Write-Ahead Logging (WAL) and recovery protocols, ensuring that transactions can be completed or recovered even in the presence of failures.

3.2.2 SLOG

SLOG [3] is a transactional key-value store designed to achieve strict serializability, low-latency reads and writes, and high throughput for local transactions. In SLOG, a region consists of servers interconnected by a low-latency network, typically within a single data center. The system adopts a master-based architecture, where each data item has a designated "home" region, ensuring that writes and linearizable reads are directed to the master replica. Transactions are classified as either single-home when all accessed data resides in one region, eliminating the need for global coordination, or multi-home, when data spans multiple regions, requiring global coordination that increases latency.

To achieve strict serializability, SLOG employs deterministic transaction execution. Transactions are pre-scheduled during a coordination phase, which establishes a global execution order outside transactional boundaries. This approach prevents runtime delays caused by cross-region conflicts and eliminates distributed deadlocks. However, this determinism imposes two specific requirements: the entire transaction must be present during the planning phase, where its execution order is determined, and its data access set must be known in advance.

Each region maintains a local log of transactions involving data it masters. These logs are exchanged between regions in batches, alongside sequence numbers to ensure that no batches are missed. This enables regions to reconstruct the local logs of others and guarantees a consistent ordering of transactions, interleaving them and creating a global log. Granules, which represent the smallest unit of data ownership in SLOG, can encompass either individual records or sorted ranges of records. To efficiently map granules to their home regions, SLOG employs a distributed index called Lookup Master, which also tracks how many times a granule's master region has changed. When a client submits a transaction, it is sent to the nearest region, regardless of where the data resides. The region consults the Lookup Master to identify the home regions for the accessed granules.

For single-home transactions, the transaction is forwarded to the corresponding home region, where it is appended to the local log to ensure consistency. Multi-home transactions, which span multiple regions, are sent to a designated region, that acts as a sequencer of the system, for coordination. This designated region determines a global execution order for the transaction by consulting the Lookup Master to identify the home regions of the involved data granules. It then generates a special *LockOnlyTxn* transaction for each home region involved. These transactions acquire locks on the relevant data granules in the correct order and are appended to the respective local logs. This ensures consistent ordering across all regions. Transactions are executed in the order defined by each region's global log. While the global log is not totally ordered across regions, all executions are equivalent, strictly serializable ones.

The reliance on a central sequencer for multi-home transactions can increase latency when the sequencer is geographically distant from the other partitions. Additionally, this sequencer also creates a potential bottleneck if many multi-home transactions require global ordering simultaneously, especially when network latency between regions is high.

3.2.3 Detock

Detock [13] presents a graph-based concurrency control approach tailored for geo-partitioned databases, enabling independent and deterministic scheduling of single-home and multi-home transactions within each region. Unlike systems such as SLOG that rely on a global ordering mechanism, Detock embeds this coordination within the asynchronous exchange and replay of locally replicated logs. Each data item is assigned a home region which holds the primary copy of the data. Clients send transactions to their nearest region, which becomes the coordinator, responsible for determining the transaction's read and write sets and annotating them with home region details via a Home Directory, analogous to SLOG's Lookup Master.

Detock relies on dependency graphs for transaction execution. When a transaction is received by its home region, it is appended to a local log. Regions asynchronously exchange their local logs so that each region eventually receives the complete logs from all other regions, reconstructing a global log. From this, a dependency graph is constructed, allowing transactions to be executed in parallel while respecting sequential log order. This eliminates the need for global coordination, guaranteeing that transactions, regardless of type, require only a single round-trip between the initiating and participating regions, while also ensuring strict serializability.

For single-home transactions, the dependency graph forms a Directed Acyclic Graph (DAG) due to strict local ordering, allowing transactions to be executed once no incoming edges remain. For multi-home transactions, each region generates a special kind of transaction based on the parts of the transaction that involve its local data. These are appended to the local logs of the relevant regions, with their initial order determined independently by each region. This independent ordering may result in deadlocks, delaying their own execution and other conflicting single and multi-home transactions. Deadlocks caused by inconsistent ordering are resolved locally and deterministically, without aborting transactions or requiring cross-region communication. Although initial dependency graphs across regions may differ due to log interleaving, they eventually converge to a consistent structure through deterministic conflict resolution.

Detock's performance suffers in skewed scenarios, where specific data items are accessed disproportionally, increasing the likelihood of deadlocks. While these are resolved deterministically, frequent conflicts can delay transaction processing. Additionally, asynchronous log replication and independent ordering may exacerbate contention in high-conflict workloads, further affecting performance.

3.3 Tree-Based Architectures

This section examines systems that, while most of them offer weaker guarantees than the previously discussed systems, introduce interesting architectural approaches to handling transactions. Unlike systems that enforce a strict global total order, these systems prioritize scalability and efficiency, frequently relaxing consistency guarantees in order to achieve better performance in geo-replicated environments. These systems achieve this by using trees in their design to efficiently manage data and transaction processing. By using hierarchical structures for data partitioning or for organizing transaction processing across distributed nodes, trees reduce the complexity of operations, improve scalability and enable efficient conflict resolution, specially in geo-replicated environment.

These architectures can reduce the coordination overhead usually associated with centralized ordering, a problem observed in previous systems, enabling a more efficient transaction processing, just as it is presented bellow.

3.3.1 Eunomia

Eunomia [14] addresses the challenge of balancing throughput and visibility latency when implementing causal consistency in geo-replicated systems. Similarly to Cure [15], Eunomia allows local updates to proceed independently, without requiring prior synchronization. Cure, a system designed to provide causal consistency with high availability, relies on global stabilization procedures to track and enforce dependencies between updates. It uses vector clocks to precisely capture causal dependencies while optimizing visibility latency. Building on these ideas, Eunomia offers a more scalable approach by introducing a deferred local serialization procedure, which avoids intrusive synchronization overheads and reduces global stabilization costs.

Eunomia assumes that, in each data center, the object-space is divided into N partitions, each using a hybrid clock [16] to assign timestamps autonomously. Clients maintain a local variable tracking the largest observed timestamp during their session, representing their causal dependencies. When a new update is issued, the partition responsible assigns a timestamp as the maximum of three values: the current physical clock, the partition's last used timestamp incremented by one, and the client's timestamp incremented by one. This ensures that updates respect the causal order while avoiding synchronization overhead.

The log, central to Eunomia, tracks update IDs and their timestamps, enforcing causal consistency by maintaining update order across partitions and regions. By separating metadata, update IDs and timestamps, from actual update content, Eunomia minimizes metadata transfer overhead between data centers. This design reduces inter-data center communication costs and allows partitions to operate independently while ensuring that all updates respect causal dependencies. Each data center maintains a vector clock, called $PartitionTime$, tracking the latest updates across partitions. Periodically, Eunomia computes a $StableTime$, the minimum value from $PartitionTime$, to identify updates that are stable - those whose causal dependencies into the log are fully satisfied. Stable updates are serialized into the log, ensuring that no updates with earlier timestamps can later arrive. This process allows Eunomia to achieve consistency while maintaining high throughput and avoiding the bottlenecks associated with global stabilization.

To support geo-replication, Eunomia incorporates a receiver module in each data center. The receiver is responsible for coordinating remote updates, ensuring that all causal dependencies are applied locally before forwarding updates to the relevant local partitions. The receiver maintains two different vectors: a queue of pending updates from each remote data center and a vector tracking the latest update operations locally applied from each remote data center. Updates are processed only when their dependencies and ordering constraints are resolved, ensuring consistency across geo-replicated data centers. Eunomia employs a red-black tree [17] to efficiently manage metadata update. This data structure enables fast insertion, deletion, and in-order traversal of updates, making it well-suited for handling large numbers of updates in high-throughput environments.

By removing synchronous global stabilization from the critical path, Eunomia achieves higher throughput and lower visibility latency, particularly for local operations. However, the system still faces challenges in environments with clock skews, which can affect stabilization times.

3.3.2 Saturn

Saturn [4] is a metadata service designed for geo-replicated systems to enforce causal consistency. It maintains a constant metadata size regardless of the number of clients, servers, partitions, or data centers involved. By decoupling metadata management from data dissemination, Saturn eliminates the trade-off between throughput and visibility latency and supports genuine partial replication, enabling scalability as the number of geo-replication sites increases.

It introduces compact metadata units, called labels, which uniquely identify operations and ensure causal consistency. Each client maintains a label that tracks their causal history, updated with every operation. This label is used for both updates and migrations between data centers, ensuring causal dependencies are maintained. Labels are compact and causally consistent, which simplifies metadata propagation and management across geo-replicated environments.

Saturn's architecture is composed by the following sub-components:

- Stateless Frontends: Act as intermediaries, intercepting client requests and forwarding them to the relevant storage servers.
- Gears: Attached to every storage server, they are in charge of intercepting operations, assigning corresponding labels with timestamps that respect causality. They propagate the associated data and metadata.
- Label Sink: A central point in each data center, it asynchronously collects labels from all gears, orders them based on timestamps, and forwards them for further processing.
- Remote Proxies: Implements updates originating from other data centers, thereby ensuring the integrity of the causal order.

To propagate metadata efficiently, Saturn uses a tree-based topology for metadata dissemination. This topology is managed by distributed serializers that minimize propagation latency. Data centers form the leaves of the tree, while serializers act as internal nodes, ensuring updates are visible with minimal delay. A key innovation is the *Weighted Minimal Mismatch* metric, which optimizes serializer placement and metadata paths to reduce delays while balancing performance-critical paths.

When building the tree, Saturn chooses the location of serializers from a predefined set of potential locations, such as existing data centers. Saturn employs a heuristic algorithm to build and refine the tree topology. Starting with a basic two-node tree, the algorithm incrementally adds new nodes, selecting configurations that minimize global mismatch. This adaptability allows the topology to scale and adjust to changes in the number of data centers or shifts in workload characteristics. By limiting metadata propagation to only relevant branches of the tree, Saturn supports partial geo-replication, reducing metadata traffic and mitigating false dependencies.

Saturn achieves a balance between scalability and efficiency. Its tree-based topology reduces metadata latency while maintaining compact metadata size, and its partial replication capabilities optimize resource use.

3.3.3 ENGAGE

ENGAGE [5] is a storage system designed for partially replicated edge environments, aiming to support efficient session guarantees while addressing the challenges of low visibility times and bandwidth constraints. The system operates on a network of cloudlets, which are small-scale edge servers distributed geographically. These cloudlets replicate frequently accessed data to reduce latency for edge applications. Unlike traditional data centers, cloudlets are designed to operate closer to users, providing fast and localized data access while supporting partial replication to conserve resources.

ENGAGE combines vector clocks and a distributed metadata service to enforce causal consistency and provide efficient session guarantees. It relies on a causally ordered log to ensure that operations dependent on each other are applied in the correct order, enabling low-latency updates. Each cloudlet tracks updates using two types of vector clocks: one capturing the causal history of each replicated object and another tracking the highest sequence numbers of updates received from remote cloudlets.

ENGAGE also introduces a hierarchical tree architecture to efficiently propagate metadata and update payloads. Leaf nodes in the tree correspond to the cloudlets, which serve client requests and synchronize updates. Inner nodes, deployed in data centers or larger cloudlets with more resources, aggregate metadata and optimize its dissemination, reducing communication delays and bandwidth usage. This tree architecture helps synchronize the log across all cloudlets, ensuring that updates are applied in the correct causal order and that dependencies between updates are respected.

Clients in ENGAGE interact with their preferred cloudlet, which processes their operations locally whenever possible. For read and write operations, clients specify session guarantees such as Read Your Writes or Monotonic Reads. Additionally, clients also maintain two different vector clocks: one for updates the client has read and another for write operations they have performed. These clocks ensure that operations are executed consistently, even when clients migrate between cloudlets. Write operations involve assigning sequence numbers and updating metadata, while ensuring that all causal dependencies are resolved before updates are applied. In order to handle remote updates, each cloudlet maintains a list of pending updates. Updates are applied only when their dependencies have been satisfied, ensuring consistency across replicas without requiring excessive synchronization. Metadata dissemination is further optimized by combining update notifications with metadata flush messages, reducing signaling overhead by piggybacking vector clock updates onto other messages.

By combining vector clocks, session guarantees, and an efficient metadata service, ENGAGE achieves low visibility times and efficient bandwidth usage in edge environments.

3.3.4 A Distributed and Hierarchical Architecture for Deferred Validation of Transactions in Key-Value Stores

The proposed system [18] combines a hierarchical communication pattern, used in [19], with message batching techniques to build a transactional system capable of supporting more clients and achieve higher throughput. It seeks to mitigate bottlenecks caused by the need for total ordering transactions, assuming an optimistic concurrency control mechanism that performs best when transaction conflicts are rare.

Two concurrency control mechanisms are explored: one using timestamps, enforcing serializability by assigning commit timestamps, and another based on acquiring locks on the accessed objects during validation, aborting transactions in case of conflicts. Transactions begin locally on the client side without immediate communication with data nodes. On the one hand, read transactions operate on a global snapshot determined by the first read operation of the client, returning the latest object version consistent with the snapshot. On the other hand, write transactions are cached locally and only sent to data nodes during the commit phase. At this point, new writes are saved as tentative versions, validated either locally or hierarchically, in the case of distributed transactions, and subsequently committed or aborted, depending on the validation outcome.

The architecture is designed for distributed key-value storage systems, where data is partitioned across multiple data nodes. Validation nodes are organized hierarchically, with leaf nodes co-located with data nodes to handle local transactions. Higher-level nodes aggregate partial validation results for distributed transactions, forming a tree structure. This hierarchy ensures scalability by limiting the scope of communication and validation based on transaction locality, minimizing latency for transactions affecting only a small subset of nodes.

For local transactions, validation is performed at the leaf node co-located with the relevant data node, ensuring low-latency processing without requiring coordination with other nodes. Distributed transactions are validated hierarchically: leaf nodes perform partial validation and propagate results to higher-level nodes, which aggregate these results and perform full validation. This approach assumes high locality, where most transactions involve a small number of nodes, reducing the number of transactions requiring higher-level validation and improving throughput.

To further enhance performance, the system employs message batching, reducing the frequency of network communication and CPU overhead. By grouping multiple messages into a single transmission, batching minimizes the overhead associated with network protocols and improves overall efficiency. This distinguishes the proposed system from earlier architectures like [19], by enabling it to support more clients and achieve higher throughput while maintaining low latency.

3.4 Discussion

In this section, we present a comprehensive discussion of the systems previously presented. First, we will discuss these systems along three key dimensions: shared (totally ordered) logs, locality in distributed logs, and tree-based architectures. By maintaining this alignment, we aim to evaluate the strengths and limitations of the systems, highlighting their trade-offs and implications for geo-distributed environments. Finally, we will provide an overall discussion to synthesize insights from these comparisons and identify areas for improvement.

3.4.1 Shared (Totally Ordered) Logs

Shared (totally ordered) logs provide strong consistency, making them an attractive abstraction for distributed systems.

Systems like CORFU and vCorfu rely on sequencers to ensure a total order of operations. While this approach is straightforward, it faces significant bottlenecks as the system size grows, with sequencers struggling to match increasing I/O bandwidth demands. Moreover, sequencers act as a single point of failure, limiting overall throughput and scalability. vCorfu enhances CORFU's design by adding transaction support, which increases its utility for applications requiring atomic operations. On the other hand, Scalog reduces the load on the centralized sequencer by aggregating information regarding updates via a hierarchical tree of aggregators, reducing bottlenecks and improving scalability. However, traversing the tree and aggregating updates introduces significant latency overheads.

More importantly, achieving a total order is inherently expensive in distributed systems. These solutions don't scale effectively, making them better suited for small clusters where the trade-offs between consistency, complexity, and scalability are less pronounced.

3.4.2 Locality in Distributed Logs

Scaling shared logs in distributed systems often involves relaxing strict total order guarantees to reduce coordination overhead. For instance, SLOG and Detock enforce a global total ordering only on a sub-type of transactions, whereas FuzzyLog focuses on serializability with more relaxed consistency guarantees.

FuzzyLog introduces the use of a directed acyclic graph (DAG) structure to maintain partial order, allowing single-home transactions to correspond to single-color updates and multi-home transactions to span multiple colors. Similarly, Detock uses a DAG for concurrency control but faces challenges such as deadlocks in skewed workloads due to its optimistic local ordering. SLOG, while leveraging a sequencer for global ordering, struggles in geo-replicated settings where high latency and scalability demands intensify bottlenecks caused by centralized coordination.

3.4.3 Tree-Based Architectures

Tree-based architectures present an effective alternative to address the challenges faced by systems like SLOG and Detock, especially in geo-distributed settings.

Systems such as Saturn, Eunomia and ENGAGE adopt tree-based designs to propagate metadata efficiently, enabling scalability while maintaining relaxed consistency guarantees. For example, Saturn and Eunomia provide causal consistency while ENGAGE supports session guarantees. However, none of these systems support transactions, limiting their applicability for workloads requiring strong consistency. In contrast, [18] employs a tree structure specifically for validating and committing transactions, enforcing serializability. By handling single-partition transactions locally and validating multi-partition transactions hierarchically, this approach provides stricter guarantees than other tree-based systems.

Despite their different goals, all these systems share the common goal of optimizing transaction processing and metadata propagation in geo-replicated environments. By leveraging hierarchical designs, they effectively distributed validation and coordination, reducing bottlenecks often associated with centralized mechanisms.

3.4.4 Overall Assessment

Table 1 outlines the key characteristics of the state-of-the-art systems previously discussed. We focus on three different aspects:

- **Consistency Model:** The consistency model is a critical aspect that determines the guarantees a system provides regarding the order and visibility of updates. Systems with strong consistency (e.g., CORFU, Scalog and SLOG) ensure serializability but often face significant coordination overhead in geo-distributed settings. In contrast, systems with weaker consistency models (e.g.,

Systems	Consistency Model	Transactions	Coordination
CORFU [1]	Strong	✗	Sequencer
Scalog [10]	Strong	✗	Sequencer w/Aggregators
vCorfu [2]	Strong	✓	Sequencer
FuzzyLog [11]	Strong	✓	Skeen(Client-Based)
SLOG [3]	Strong	✓	Sequencer
Detock [13]	Strong	✓	Optimistic
Eunomia [14]	Weak	✗	Tree
Saturn [4]	Weak	✗	Tree
ENGAGE [5]	Weak	✗	Tree
[18]	Strong	✓	Tree

Table 1: Comparison between systems presented in the Related Work section

Eunomia, Saturn and ENGAGE) prioritize scalability and efficiency but may limit applicability for workloads that require strict guarantees.

- **Transaction Support:** Transaction support is essential for systems that need to execute operations atomically across multiple partitions. Systems such as vCorfu, SLOG and Detock include robust transactional mechanisms, enabling them to handle operations that span multiple nodes. However, systems such as CORFU, Scalog and Saturn do not support transactions, which limits their applicability in use cases requiring atomicity.

- **Coordination Mechanism:** The coordination mechanism determines how a system maintains order and consistency. Sequencer-based systems, such as CORFU and vCorfu, rely on centralized components to enforce total order. While this simplifies the coordination process, it introduces significant bottlenecks in scalability due to the reliance on a single point of failure. Scalog mitigates this scalability bottleneck by aggregating information regarding updates, but incurs a large latency overhead in the process. Fuzzylog adopts a client-based mechanism using Skeen's algorithm, which eliminates centralized coordination but can introduce latency if the client is geographically distant from participating nodes. Optimistic approaches, like the one used in Detock, allow the system to locally order multi-home transactions in each region assuming convergence without immediate synchronization. While this reduces the coordination overhead, it introduces potential challenges in resolving conflicts, especially under skewed workloads. Finally, tree-based mechanisms leverage hierarchical designs to propagate metadata efficiently, reducing coordination overhead and enabling scalability.

4 Coordinator-Based Total Order for Geo-Distributed Transactions

With our proposed architecture, we aim to address some of the limitations of previous architectures, particularly in systems like those presented in [3, 13], when supporting transactional execution. As previously discussed, sequencer-based solutions have some limitations, primarily the fact that all the transactions have to pass through a specific location, which isn't an optimal solution, specially in cases where participants can potentially be geographically distant from the sequencers, as it increases communication costs and latency. In addition, some systems also struggle to handle skewed workloads, where certain regions experience disproportionate transaction loads.

The hierarchical structures of tree-based architectures offer a promising solution to solve these limitations by creating a hierarchical structure that helps coordinate and order transactions without the need to traverse to a central point. However, traditional tree-based solutions still face bottlenecks, particularly when managing multi-home transactions. A critical observation in these architectures, which serves as the main inspiration for our proposed solution, lies in how nodes at the edges of the tree (i.e., those at the far right and far left of the hierarchy) often appear distant from each other within the architecture. The perceived distance forces transactions involving these nodes to traverse the entire tree, even though they may be geographically close in reality, adding unnecessary latency and overhead.

In this section, we present our proposed architecture and the motivation behind its design. We detail how transactions are processed within this architecture and introduce a proposed algorithm to enhance its deployment, focusing on mapping the architecture to its physical placement across geo-locations.

4.1 Coordinator-Based Architecture for Total Order

Our proposed architecture supports a transactional key-value store (KVS), partitioned across a set of geo-replicated locations. It ensures a total order of transactions, enabling the implementation of strict serializability as well as other consistency levels that rely on total order, enabling reliable coordination and execution across geo-distributed locations.

Each location, similar to systems such as [3, 13], is the "home" location of a specific object. The architecture differentiates between single-home transactions and multi-home transactions. On the one hand, single-home transactions, which involve only one location, can be immediately committed to the corresponding location without additional coordination. On the other hand, multi-home transactions, which involve multiple locations, require coordination to achieve a global total order for that transaction. Additionally, every location maintains a local log that records transactions that affect its corresponding object, ensuring a total order of transactions through the use of Paxos [20] for consensus. This approach guarantees that updates are consistently applied across all replicas within the location.

A core aspect of our architecture is the use of Skeen's algorithm to achieve a total order for multi-home transactions. The algorithm operates by having processes propose timestamps for events and then agreeing on the maximum proposed timestamp as the final order. In the context of our architecture, Skeen's algorithm enables geo-locations to propose timestamps for multi-home transactions and ensures that these timestamps are aggregated to determine a globally consistent total order.

However, implementing this algorithm presents specific challenges. For instance, if geo-locations directly communicated with each other to exchange and determine the maximum timestamp, the communication cost would grow quadratically with the number of participants. Alternatively, similarly to [11], delegating the responsibility for aggregating timestamps to the client would introduce its own drawbacks: clients may be geographically distant from the locations involved, potentially leading to higher latency. This limitation is similar to the one faced by sequencer-based systems.

To address these challenges, we use a set of coordinators - dedicated entities responsible for aggregating the proposed timestamps. Coordinators are designed to minimize communication overhead and latency by acting as central aggregation points for each combination of geo-locations involved in a transaction. For instance, in a system with n geo-locations, a coordinator is assigned to transactions involving all n locations. Additional coordinators manage subsets of locations, such as combinations of $n-1$ locations (with a total of $\binom{n}{n-1}$ such coordinators), combinations of $n-2$ locations (with $\binom{n}{n-2}$ coordinators), and so forth, down to combinations involving two geo-locations. This eliminates the need for direct communication among geo-locations or reliance on distant clients, ensuring efficient execution of the algorithm.

Figure 1 provides a logical representation of the proposed architecture in a system composed of three distinct geo-locations. Each geo-location is responsible for managing a specific object ("A", "B" or "C"). The figure also illustrates the coordinators and their connections, highlighting how they interact. For example, a multi-home transaction involving geo-locations "A", "B" and "C" is assigned to the coordinator "ABC", which is responsible for determining a total order for transactions involving that combination of geo-locations.

In the context of our architecture, Skeen's algorithm is executed collaboratively by both geo-locations and coordinators. Geo-locations propose timestamps, while the coordinators aggregate these proposals to determine the total order of multi-home transactions, with the final order being the result of this

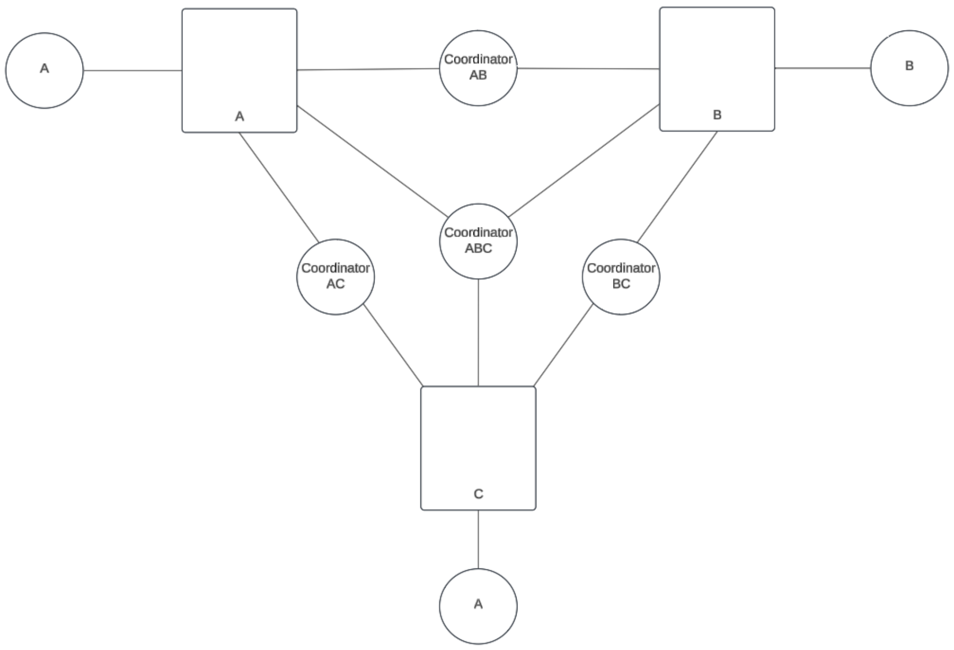

Figure 1: Logical Representation

coordinated process. These coordinators efficiently bridge the gap between the algorithm's requirements and the practical constraints of geo-distributed systems.

At this stage, we assume that fault tolerance will be handled using an approach similar to CORFU and NoPaxos [21]. In the case of a failure, the entire system stops momentarily to allow for reconfiguration, ensuring that all participants agree on and are made aware of the updated configuration before resuming operation.

4.2 Transaction Handling

Transactions in this system are classified into single-home transactions and multi-home transactions, with the handling process differing based on the type. Each transaction starts at the corresponding geo-locations, which determine its nature and act accordingly.

4.2.1 Single-Home Transactions

Single-home transactions, as specified before, involve only one geo-location and can be processed and committed locally without requiring communication with other geo-locations or coordinators.

1. **Client Begins Transaction:** The client submits the transaction to the relevant geo-location. For instance, if the transaction involves the object "A", the client sends the transaction to the geo-location responsible for such object.

2. **Local Check and Commit:** The geo-location determines that the transaction is single-home, as it only affects its managed objects. Since no coordination is required, the transaction is directly inserted into the geo-location's local log, which maintains a totally ordered sequence of transactions using a Paxos based mechanism. This minimizes latency, allowing for fast and efficient processing of localized operations.

4.2.2 Multi-Home Transactions

Multi-home transactions span multiple geo-locations and require coordination to ensure a globally consistent total order.

1. **Client Begins Transaction:** The client submits the transaction to all the geo-locations involved in the transaction.
2. **Local Check and Forwarding to Coordinator:** Each geo-location checks whether the transaction affects other geo-locations. If it does, the transaction is classified as multi-home, and coordination with a coordinator is required. Each participating geo-location assigns a local timestamp to the transaction, ensuring it is larger than any previously attributed to any transaction at that location, representing its view of the transaction's order.
3. **Coordination via Skeen's Algorithm:** The geo-locations and the coordinator responsible for the combination of geo-locations (e.g., "AB" for "A" and "B") execute Skeen's algorithm collaboratively. Each geo-location sends its proposed timestamps to the coordinator, which aggregates the timestamps and determines the global order by taking the maximum. This process ensures a total order across all participating geo-locations while minimizing communication overhead.
4. **Commit Transaction:** Once the global timestamp is established, the transaction is added to a queue at each participating geo-location. The transaction waits to reach the top of the queue, at which point the geo-locations use Paxos to insert the transaction into their local logs in the globally agreed order.

4.3 Coordinator Placement Optimization

Just as the algorithm introduced in [4], we aim to develop an algorithm that assigns physical locations to coordinators in a way that minimizes latency while balancing the system's load. The primary goal is to strategically place coordinators while accounting for both communication efficiency and workload distribution.

We assume that we have information about the load distribution, specifically the percentage of transactions affecting each combination of objects. This information will help us guide the placement of coordinators to their physical locations. For example, if the load distribution reveals that a coordinator has 0% load, meaning there are no transactions for the associated combination of locations, that coordinator can be eliminated. This step simplifies the system and ensures resources are allocated only to necessary coordinators.

Each home region is located in a distinct geo-location, specifically in different data centers, which are the only data centers utilized in the system. To minimize latency, coordinators should be placed as centrally as possible, considering the geo-locations of the home regions they interact with. However, central placement alone is insufficient to ensure scalability, knowing the workload distribution is crucial. For instance, we aim to prevent situations where a single geo-location is overloaded with multiple high-demand coordinators. By understanding the percentage of transactions affecting each coordinator, the algorithm can evenly distribute coordinators across machines, preventing bottlenecks and ensuring optimal resource utilization.

Figure 2 illustrates a possible deployment of the algorithm, showing the physical placement of coordinators across geo-locations. Each geo-location is represented with a distinct color, emphasizing the distribution of coordinators and their roles in transaction coordination.

By addressing both latency and load distribution, the algorithm ensures that coordinators are placed to support efficient transaction coordination and scalability, even under diverse workloads. Once the optimized placement is determined, the system applies the configuration directly, deploying coordinators to the most suitable geo-location.

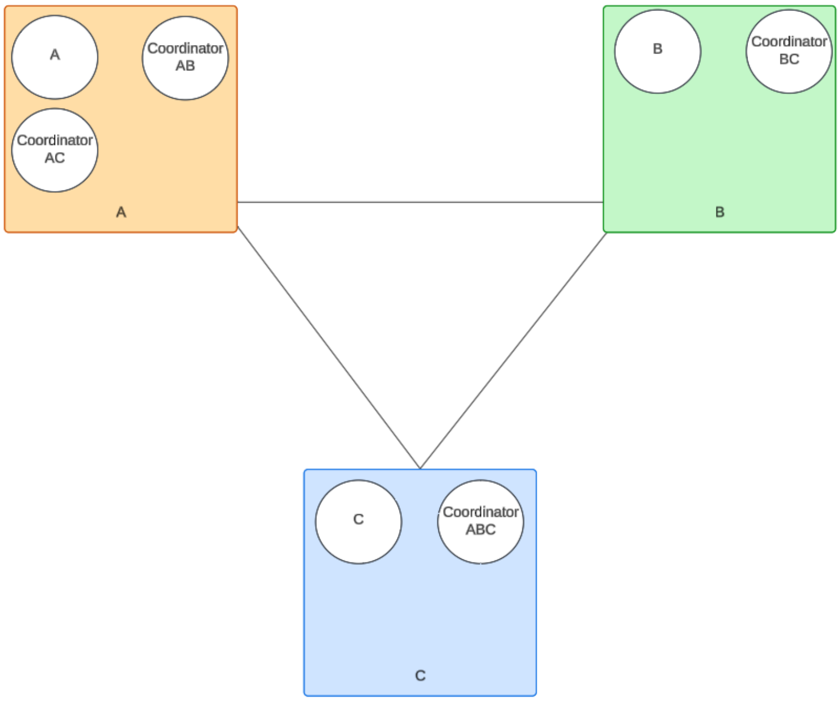

Figure 2: Physical Representation

5 Evaluation

To evaluate the performance of our proposed architecture, we will utilize the codebase available from Detock[1] and SLOG[2], the systems discussed in Section 3.2. This codebase will be used to deploy our architecture, enabling a direct comparison of our system against these approaches.

To assess the performance of our system, we will test it under varying workloads, similar to those used in Detock and SLOG. By adjusting the percentage of single-home and multi-home transactions, we will evaluate how our architecture performs in scenarios dominated by single-home transactions versus those with more multi-home transactions. This approach will help us identify the scenarios in which our system excels and understand the impact of workload composition in its efficiency.

Our evaluation will focus on several properties:

- **Latency:** Measure the latency of transactional operations across different workloads. This includes varying the number of objects accessed per transaction and the proportion of "home" versus "multi-home" (distributed) transactions.
- **Throughput:** Measure the throughput of transactional operations across different workloads, assessing its ability to handle high volumes of transactions efficiently.
- **Scalability:** Analyze how performance is impacted as the system size increases, considering both the number of nodes and transaction complexity.

By focusing on these properties, we expect to gain a better understanding of the trade-offs and limitations introduced by our design.

[1]The source code is available at https://github.com/umd-dslam/Detock
[2]The source code is available at https://github.com/umd-dslam/Slog

6 Work Schedule

Future work is scheduled as shown in Figure 3.

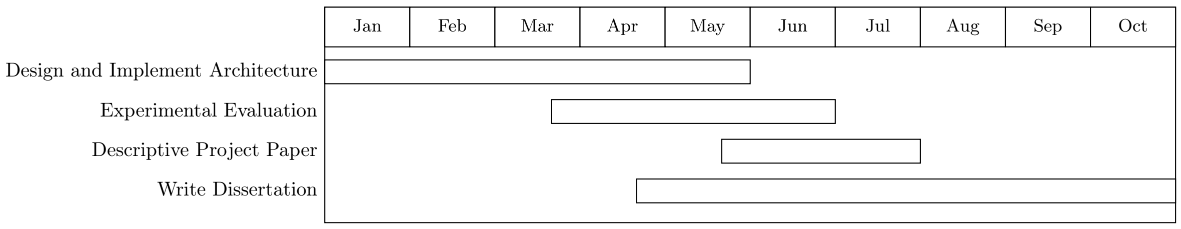

Figure 3: Planned Schedule

7 Conclusion

Geo-distributed environments present unique challenges for transaction execution, particularly in balancing efficiency, scalability and consistency. Traditional sequencer-based approaches often face high latency and bottlenecks due to centralized coordination, while existing tree-based architectures lack efficient mechanisms to support transactions across diverse workloads.

This report has presented a system designed to address the challenges of transaction execution in geo-distributed environments. By using a set of independent coordinators, our architecture aims to optimize transaction coordination while minimizing communication overhead.

In summary, we have explored previous works to identify their strengths and limitations, using these insights, we proposed a solution and detailed its implementation plan. Finally, we outlined our evaluation methodology and presented a schedule for future work.

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

Bibliography

[1] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," in *Association for Computing Machinery*, 2013.

[2] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson *et al.*, "{vCorfu}: A {Cloud-Scale} object store on a shared log," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 35–49.

[3] K. Ren, D. Li, and D. J. Abadi, "Slog: Serializable, low-latency, geo-replicated transactions," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, 2019.

[4] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 111–126.

[5] M. Belém, P. Fouto, T. Lykhenko, J. Leitão, N. Preguiça, and L. Rodrigues, "Engage: Session guarantees for the edge," in *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2022, pp. 1–10.

[6] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, 1978.

[8] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 1994, pp. 140–149.

[9] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–34, 2016.

[10] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. Van Renesse, "Scalog: Seamless reconfiguration and total order in a scalable shared log," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 325–338.

[11] 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)*, 2018, pp. 357–372.

[12] 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.

[13] C. D. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023.

[14] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive deferred update stabilization for efficient {Geo-Replication}," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 83–95.

[15] 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)*. IEEE, 2016, pp. 405–414.

[16] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer, 2014, pp. 17–32.

[17] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE, 1978, pp. 8–21.

[18] J. B. S. Amaro, "A distributed and hierarchical architecture for deferred validation of transactions in key-value stores," 2018.

[19] J. Grov and P. C. Ölveczky, "Scalable and fully consistent transactions in the cloud through hierarchical validation," in *International Conference on Data Management in Cloud, Grid and P2P Systems*. Springer, 2013, pp. 26–38.

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

[21] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say {NO} to paxos overhead: Replacing consensus with network ordering," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 467–483.