DIDA-TKVS
Design and Implementation of Distributed Applications 2024-2025

Martim Mendes
ist1 102932
martimccmendes@tecnico.ulisboa.pt

João Norte
ist1 102795
joaonorte@tecnico.ulisboa.pt

Pedro Freitas
ist1 103168
pedrogmfreitas@tecico.ulisboa.pt

Abstract

This report presents the design and implementation of a distributed transactional key-value store (dida-tkvs) using the Simple Paxos consensus algorithm. Our approach ensures consistency and fault tolerance across multiple servers by coordinating client transactions through Paxos consensus rounds. The architecture includes key components such as RequestArchiveStore, responsible for storing and handling client requests and PaxosLog, responsible for keeping track of the agreed order to process the requests. Our solution abstracts components to limit their responsibilities, ensuring that each element interacts only with the data it requires. Clients send transactional requests to the servers, and each server can act in various roles: as a learner, an acceptor, or as both, with the proposer role assigned only to the leader. To ensure consistency and concurrency control, each server maintains its own PaxosLog and a RequestArchiveStore. By structuring the system this way, we streamline Paxos coordination across servers, reduce the complexity of managing concurrent operations, and achieve fault-tolerant, efficient replication of the key-value store across multiple nodes.

1. Introduction

The dida-tkvs project aims to develop a distributed, transactional key-value store that uses Paxos consensus to ensure consistency across multiple servers. In distributed systems, achieving consensus on updates in the presence of failures or concurrency is challenging. Paxos provides a robust solution by allowing multiple replicas to agree on a sequence of transactions, ensuring data integrity even when some nodes fail or operate concurrently.

Each server plays one or more roles: learner, acceptor, and proposer (leader); depending on its configuration. The clients send transactional requests to the servers, and these requests are stored as RequestArchives in a RequestArchiveStore for tracking and handling purposes. The servers coordinate using Simple Paxos to propose, accept, and commit updates, with each proposal taking the form of a PaxosProposal structure that tracks timestamps and commit status and then being logged in PaxosLog when consensus is achieved.

Reconfiguration is supported through Paxos rounds, allowing the system to dynamically switch between different server configurations. We also implemented several debug modes, such as server crashes, freezes, and slow modes, to test the behavior of the system under various failure scenarios. These debug modes help simulate real-world issues, ensuring the robustness and fault tolerance of the system. Our solution simplifies thread coordination, ensuring each component has controlled access, thus preventing race conditions and promoting efficient Paxos execution.

2. System Architecture

Our system architecture is centered around a distributed key-value store replicated using Simple Paxos, where each server manages client requests and participates in consensus protocols. The core components are:

RequestArchive and RequestArchiveStore: These store client requests. Each server maintains its own archive, with requests marked as proposable or unproposable based on their status in the Paxos process.

- **PaxosProposal and PaxosLog**: Each server tracks proposed and committed requests in its PaxosLog, ensuring that requests are executed in the correct order once consensus is achieved.

- **PaxosParticipant**: This class serves as the foundation for the server roles (learner, acceptor, and proposer), promoting modularity and reuse. Each server runs instances of these roles, coordinating as needed during Paxos rounds.
- **PaxosLearner, PaxosAcceptor, and PaxosProposer**: Servers act as learners to process client requests, acceptors to participate in Paxos rounds, and proposers to lead the consensus process when elected as leader. The proposer handles the key task of initiating Paxos rounds and driving reconfiguration when necessary.

This architecture simplifies thread coordination and ensures each component has isolated access to the data it needs, improving fault tolerance and ensuring the correct replication of the key-value store.

2.1. RequestArchive

The RequestArchive is the data structure used by the server to store client requests. It includes a *responseObserver* for responding to the client, a *requestId*, and a boolean flag called *proposable*. Only requests marked as proposable are eligible for proposal by the Paxos leader. Once a consensus is reached for a request during a Paxos round, it is marked as unproposable, ensuring that the same request cannot be proposed multiple times.

2.2. RequestArchiveStore

The RequestArchiveStore is a data structure that maintains all the RequestArchive entries for each server. Each server manages its own instance of RequestArchiveStore, interacting with it through the following methods:

- **getRequest(Id)**: Retrieves the stored request corresponding to the given Id.
- **addRequest(RequestArchive)**: Adds a new RequestArchive entry to the store.
- **getNext()**: Returns the next request eligible for Paxos proposal (must be marked as proposable).

2.3. PaxosProposal

PaxosProposal is a data structure designed to track essential attributes of a request, including its read and write timestamps, along with its commit status. Each request is formatted as a PaxosProposal when proposed by a leader during the Paxos consensus process It includes a *requestId*, *writeTs*, *readTS* and a boolean flag called *comitted*. Only committed requests can be executed by the server, signifying that consensus on the request's order has been reached and will remain unchanged.

2.4. PaxosLog

PaxosLog is a data structure maintained by each server to log requests in the order determined by Paxos rounds. Each server tracks a counter for previously executed requests and proceeds to execute the next request in the log only if it has been committed. Servers interact with the PaxosLog through the following methods:

- **getPropose(order)**: Retrieves the proposal logged at the specified order.
- **setPropose(order, PaxosProposal)**: Logs a new proposal at the designated order.
- **commitPropose(order, PaxosProposal)**: Commits the proposal for the specified order, indicating that consensus has been achieved.
- **getCommittedRequestId(order)**: Retrieves the current proposal, which must exist and be committed.

2.5. PaxosParticipant

In our implementation, we designed a class hierarchy centered around the PaxosParticipant class, which serves as a foundational component for different roles in the Paxos protocol: learner, acceptor, and proposer. This design promotes code reuse and modularity while allowing for clear separation of responsibilities among the different roles. Each server keeps an instance of learner, aceptor and proposer and executes code from each when necessary. The PaxosParticipant class contains essential references to three key components: **RequestArchiveStore**, **PaxosLog** and **ServerState**.

2.6. PaxosLearner

Each server essentially functions as a PaxosLearner, which is responsible for managing the main flow of client requests. As requests arrive, Learners store them in the RequestArchiveStore for tracking and processing. They handle read operations and execute transactions, ensuring the integrity of the data. Additionally, Learners receive learn requests, committing the appropriate proposals, which allows these proposals to be executed later.

2.7. PaxosAceptor

PaxosAceptors are resposible for participating in paxos rounds. Upon receiving a *prepare* (phase one request) they should check if the read timestamp is the same of higher than the request timestamp, and if it is respond with a promise including the last proposal they have received for

that specific order (empty answer if none). Additionally they must update the read timestamp for that proposal in the PaxosLog.

When they receive an *accept* (phase two request), they should accept it only if the read timestamp is the same of higher than the request timestamp. After acceptance, they must update both the read and write timestamps. Following this, they should send learn requests to all learners to propagate the accepted proposal.

2.8. PaxosProposer

The PaxosProposer is responsible for initiating and coordinating Paxos rounds to achieve consensus among servers. It manages the process of proposing values and ensures they are accepted by a majority of acceptors.

When the Proposer becomes the leader it checks if there are pending requests in the RequestArchiveStore. If it is the leader and there are requests to propose, the Proposer starts a Paxos round by executing Phase One.

In Phase One (prepare), the Proposer sends a request to all acceptors, asking for a promise that they will not accept any proposals with a lower timestamp. If the majority of acceptors respond positively, the Proposer proceeds to Phase Two (accept), where it either proposes its own new value or a value that was already partially accepted by the acceptors.

If any phase fails, the Proposer increments its ballot number and tries again. If not then it marks the proposed request as Unproposable and moves to the next Paxos round. The Proposer also handles reconfigurations, checking if a proposed request involves a configuration change, and updating the system configuration if necessary.

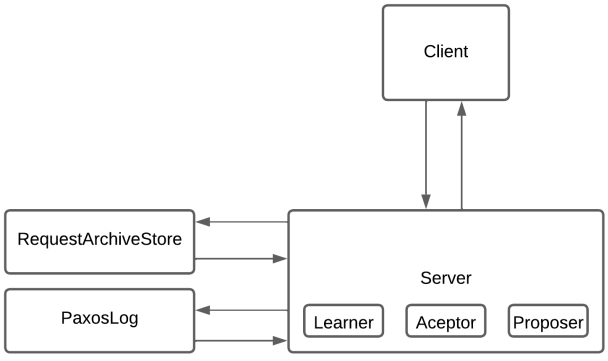

Figure 1. System architecture representation.

3. Protocol (Main Flow)

The main flow of the system begins with the client sending a request to all servers. Each server manages its own RequestArchiveStore and PaxosLog. Upon receiving a request, a server stores it in its RequestArchiveStore using the RequestArchive format.

Proposers become active in two scenarios: when they are elected as leaders through a ”leader on” command from the console, or when they receive a client request. If the proposer is the current leader, it initiates a new Paxos round. The first leader to act skips the initial step of Paxos, while any subsequent leader must begin by executing step one.

In step one, the proposer sends a prepare request to all acceptors in the current configuration, targeting a specific index in their PaxosLog. It waits for a majority of responses. If any response is a ”nack” (indicating that the proposal was rejected due to a higher ballot number), the leader increases its ballot number and retries the process. Acceptors handle the prepare request by responding with a promise that includes the last written value for that index in their PaxosLog, or they return ”-1” if no value exists.

Once the proposer receives a majority of responses, it either chooses a new value if all answers were empty or selects the value with the highest write timestamp from the responses. The proposer then proceeds to step two.

In step two, the proposer sends an accept request to all acceptors and again waits for a majority. If any acceptor rejects the request (again with a ”nack”), the leader increases its ballot number and retries the process from step one. Acceptors that receive the accept request will validate the timestamps, and if valid, they respond with an ”accepted” message to the proposer. They also send a learn request to all learners once they have accepted a value, signaling that a consensus has been reached for that index.

When the proposer receives a majority of accepted responses, consensus is achieved. At this point, the proposer marks the corresponding request in the RequestArchive as unproposable, preventing it from being proposed again. If the request is related to reconfiguration, the proposer handles it accordingly. After successfully completing the Paxos round, the leader checks whether it can start another round, depending on whether it has more requests and is still the active leader.

Learners play a critical role in finalizing the consensus. When a learner receives a majority of learn requests for the same PaxosLog index, it commits the PaxosProposal at that index in its PaxosLog. Additionally, when a learner stores a request in its RequestArchive or commits a PaxosProposal, it attempts to execute the corresponding transaction. However, this can only occur if the PaxosProposal has been committed in the PaxosLog, the request is stored in the RequestArchiveStore, and all preceding requests in

the PaxosLog have already been executed.

3.1. Learn Majority

The *LearnCounter* class is used to monitor the number of "learn" messages received for each Paxos proposal, ensuring that a majority of acceptors agree on a value before it can be committed. For each Paxos round or order, the class maintains a counter associated with the current leader handling that round.

When a server receives a learn message, it increments the counter for the corresponding proposal. This counter is linked to the leader's ID, and if the leader changes, the counter is reset to ensure accurate tracking. Once the counter reaches a majority, meaning more than half of the acceptors have agreed, the proposal can be considered committed. This mechanism helps ensure that consensus is achieved before moving forward with the execution of any transaction.

3.2. PaxosLog Optimization

To optimize Paxos rounds, when a proposer becomes active, it first checks the PaxosLog to identify what has already been committed, stopping at the first gap it encounters. By doing this, the proposer ensures that, if it is the leader, it will only initiate a new Paxos round for indexes in the PaxosLog that have not yet reached consensus. This approach guarantees that all previous indexes have been committed and are known by the leader, avoiding unnecessary rounds and ensuring the consistency of previous decisions before proceeding with new ones.

4. Reconfiguration

Our algorithm ensures that a leader can only begin a Paxos round 'i' if all prior rounds have been committed in the PaxosLog. After successfully completing each Paxos round, we verify whether the request involves writing to key 0, which is reserved for configuration changes. If the request is intended to update the configuration, the system transitions to the next configuration. Since a new Paxos round can only start once the previous one has fully concluded, this guarantees that configuration changes occur seamlessly, without interrupting or overlapping with ongoing Paxos rounds. Consequently, every Paxos round following a reconfiguration (round 'i') will operate under the updated configuration, preventing any mid-round configuration changes. Additionally, before starting a new round, the leader verifies that it remains a valid leader under the updated configuration. Acceptors can trust the leader's decisions, knowing that the leader is always operating within the correct configuration, so if this acceptor was included in the requests for this proposal it means that the acceptor is a participant of the configuration of the proposal.

5. Console (Debug Modes)

In the system, each server has an instance of the *DebugModes* class, which is responsible for applying various debug modes like freeze, slow mode, and crash. The *applyDebugMode* method is called every time a request is processed, ensuring that the appropriate debug behavior is applied to the request. For example, in freeze mode, the server will block client requests until it is unfrozen, while in slow mode, the server introduces random delays before processing. The *setdebug* method, implemented in the *ConsoleServiceImpl*, allows the console to set or change these debug modes for the servers, and the corresponding actions are triggered accordingly. This structure ensures that debug behaviors are consistently enforced across the system without disrupting normal operation.

5.1. Crash

When the server receives a crash request, it terminates immediately using *System.exit()*. To handle this gracefully, we implemented a shutdown hook that ensures all running threads are properly closed. The shutdown hook is registered with *Runtime.getRuntime().addShutdownHook*, and upon termination, it forces the immediate shutdown of the gRPC server with *server.shutdownNow()*. This mechanism ensures that the server exits cleanly while closing all active gRPC server threads to prevent resource leaks or lingering processes.

5.2. Freeze

The freeze and unfreeze functionality is implemented to simulate a server being unresponsive to client requests. When a server receives a request while in the freeze mode, it enters a waiting state, effectively blocking the processing of that request. This is achieved through the *wait()* method, which pauses the execution of the thread until it is notified to resume. When the server receives an unfreeze request, it calls the unfreeze() method, which invokes *notifyAll()*. This action wakes up all threads that are currently waiting due to the freeze, allowing them to proceed with their processing. Consequently, the system transitions smoothly back to normal operation, enabling it to handle requests again.

5.3. Slow Mode

The slow mode in the provided code introduces a delay in processing client requests to simulate network latency or server slowdowns. When a request is received and the slow

mode is activated, the server randomly sleeps for a duration between a predefined minimum and maximum time. This simulates a slower response time without affecting the request's integrity or flow.

5.4. Block Phase One

The block phase one mode simulates a scenario where the server temporarily (until an unblock signal is received) stops processing only phase one requests, similar to the freeze mode but restricted to the first phase of Paxos. This is implemented using the *wait* and *notifyAll* methods.

5.5. Block Phase One and Two

The block phase one and two mode functions similarly to the freeze mode but applies exclusively to both phase one and phase two requests. This is also achieved using the *wait* and *notifyAll* methods.

6. Execution Scenarios

There were some execution scenarios we kept in mind while working on our solution, we describe a few of them in this section:

First scenario - Paxos before Request arrival: In some cases, a Paxos round may complete and reach consensus on a transaction before the corresponding client request has arrived at **all servers**. This scenario is particularly relevant when the client request is delayed, and consensus is reached without having the actual transaction details yet. From the learner's perspective, when consensus is achieved on a transaction, the result of the Paxos round is saved in the server's PaxosLog. However, since the request itself has not yet arrived, the transaction is not executed at this stage. The learner will execute the transaction only once it has both the consensus result and the actual client request details. When the delayed client request eventually arrives, the learner checks the PaxosLog for the request's position in the sequence. With the committed order and the necessary request data, the learner then processes the request, maintaining the correct execution sequence.

Second scenario - PhaseTwo before PhaseOne: At times, from the acceptor's perspective, phase two of the Paxos process might occur before phase one. This is not an issue, as the acceptor can still process phase two, since majority has definitely been reached in phase one. The acceptor checks if the timestamp in the request is greater than or equal to the read timestamp of the current proposal in PaxosLog. If so, it accepts the proposal, denying any outdated leader attempts, and updates the read and write timestamps for that proposal accordingly. When the phase one request eventually arrives, the server responds, but its response has no impact in the program execution since the leader has reached majority so it's no longer waiting on any answers.

Third scenario - Learn before PhaseOne and PhaseTwo: From the learner's perspective, it is possible to receive a learn message before any preceding Paxos messages have arrived. Servers operate with the assumption that the leader's decisions are correct, so if a majority of learn messages is received (indicating that Paxos has reached consensus), the PaxosLog is updated accordingly. If the corresponding client request is already stored in the RequestArchiveStore, the transaction can be executed immediately. This is acceptable because the system is designed to handle multiple instances of learn or accept messages without affecting the final outcome in the PaxosLog. When the delayed phase one and phase two messages eventually arrive, the server may still respond to them, but these responses have no impact, as the leader has already reached consensus and moved on, and is no longer waiting for any responses.

7. Conclusion

In conclusion, the dida-tkvs project demonstrates the implementation of a distributed transactional key-value store using Simple Paxos to ensure consistency and fault tolerance across multiple servers. Through the use of Paxos rounds, servers reach consensus on client transactions while handling reconfigurations seamlessly. Our design focuses on clear separation of responsibilities and server roles with thread coordination, ensuring correctness under various execution scenarios. The inclusion of debug modes allows for rigorous testing of the system's behavior, further validating its reliability.

References

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