Distributed Transaction Processing System using 2PC

Harshit Timmanagoudar
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
htimmanagoudar@ucsd.edu

Vibha Murthy
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
vhmurthy@ucsd.edu

I. Abstract

In this project, we implement a fault-tolerant distributed transaction system based on the Two-Phase Commit (2PC) protocol. The system consists of a central Coordinator, a set of distributed Key-Value (KV) servers, and external clients that issue transactions. The Coordinator ensures transactional atomicity and durability across multiple KV servers, even in the presence of partial failures, crashes, and network partitions. Our system supports the full 2PC flow (Prepare, Commit, Abort), implements a Termination Protocol for safe recovery, and handles edge cases such as Coordinator crashes, Server crashes, and incomplete transactions using PostPrepare logic and peer querying. Persistent logging is used to guarantee crash recovery at both the Coordinator and the KV servers. Additionally, we built an automated test framework that simulates critical failure scenarios to validate system correctness and robustness. Our implementation demonstrates that the system can correctly recover from Coordinator and Server crashes at any point in the protocol, maintaining strict transactional semantics and ensuring data consistency across the distributed KV store. Future work includes supporting concurrent transactions that do not overlap on conflicting keys to improve system throughput.

II. Introduction

Modern distributed systems often require **atomic and consistent updates across multiple nodes**, even when the underlying nodes may fail or be partitioned. This need arises in a variety of contexts, such as financial systems, cloud storage, and distributed databases. In such environments, a distributed transaction protocol ensures that a group of updates either *commit together* or *abort together*, preserving **atomicity** and **consistency** in the face of failures.

A widely used protocol for coordinating such distributed transactions is the **Two-Phase Commit (2PC)** protocol. 2PC divides the transaction process into two distinct phases: a **Prepare phase**, where all participants vote on whether they can commit the transaction, and a **Commit phase**, where the final decision is made and disseminated to all participants. The protocol is simple and effective, but presents several challenges: it is a **blocking protocol** under certain failure conditions, and achieving correct recovery in the presence of **Coordinator crashes** and **participant crashes** is non-trivial.

Our project implements a complete **distributed 2PC system** with the following architecture:

- A **Coordinator** process drives the transaction flow, coordinating with clients and KV servers.
- A set of distributed **KV servers** act as transaction participants, maintaining a simple key-value store with full transactional semantics.
- A **Client** interacts with the Coordinator to initiate transactions such as `add`, `subtract`, `transfer`, and `get balance`.

We designed the system to tolerate multiple real-world failure modes:

- Coordinator crashes at any point in the 2PC flow.
- Server crashes before or after Prepare, during Commit, or after Commit.
- Network partitions between Coordinator and participants.
- Late and duplicate messages (Prepare, Commit, Abort).
- Client retries and crashes.

Persistent **logging** at both the Coordinator and KV servers guarantees that transactions can be recovered correctly after a crash. A **Termination Protocol** ensures safe progress and consistency even when Coordinator recovery is delayed or when peers need to resolve the outcome of uncertain transactions. Additionally, our implementation supports a **PostPrepare** mechanism where participants query the Coordinator and their peers to resolve incomplete transactions after a failure.

A high-level view of our architecture is shown below:

Client → **Coordinator** → Multiple **KV Servers**
↗ **Log** ↖ ↗ **Log** ↖

- **Client**: sends transaction requests to Coordinator.
- **Coordinator**: orchestrates 2PC, maintains transaction log.
- **KV Servers**: participate in 2PC, maintain prepare/commit/abort logs, participate in PostPrepare.

Our system includes a complete **automated test suite** that simulates a variety of failure scenarios to ensure full correctness and recovery across all phases of the 2PC flow.

III. System Design and Architecture

A. Overall Architecture

Our system implements a fault-tolerant distributed transaction system using the Two-Phase Commit (2PC) protocol to coordinate operations across multiple key-value (KV) backend servers. The architecture consists of three core components:

- **Coordinator**: Manages the lifecycle of distributed transactions and drives the 2PC protocol. It handles client requests, logs transaction state transitions, and performs recovery.
- **KV Backend Servers**: Store key-value data and act as participants in 2PC. They implement the logic for `Prepare`, `Commit`, and `Abort` phases and maintain local logs for recovery.
- **Client**: Submits transactions to the system through the Coordinator, waits for the outcome, and sends acknowledgments.

Data is partitioned into bins, which are deterministically mapped to backend servers using a consistent hashing function. This ensures even distribution and scalability. Communication flows from *Client* $\rightarrow$ *Coordinator* $\rightarrow$ *KV Servers*, and ownership of keys is resolved through bin-to-backend mapping.

B. Coordinator Components

1) Transaction Lifecycle and Processing: The Coordinator exposes gRPC endpoints (`Txn`, `AckTxn`, `GetStatus`) to interact with clients. It maintains an in-memory queue of transactions and persists them to a `queue.json` file. A `QueueWorker` dequeues and drives each transaction through 2PC. Meanwhile, a `ChannelWorker` handles incoming transactions and logs them.

2) Transaction State Tracking: Each transaction is tracked in an in-memory `TxnMap` with possible states: `Pending`, `Prepared`, `Committed`, or `Aborted`. Upon restart, the Coordinator uses its persistent `txn.log` to reconstruct transaction state and resume processing.

C. KV Backend Server Behavior

1) 2PC Participation: Each KV backend implements `Prepare`, `Commit`, `Abort`, and `QueryTxnStatus` methods using gRPC. They maintain a local store and a persistent `prepare_log`, which is saved to a `kv_backend_X.log` file. This ensures they can recover in case of a crash.

2) PostPrepare Termination Protocol: After recovery, each backend launches a `PostPrepare` goroutine for unresolved transactions. This goroutine:

- Checks the local log.
- Queries the Coordinator.
- Queries peer backends.

If the transaction outcome cannot be determined, the backend discards the prepared state to avoid indefinite blocking.

D. Persistent Logging

- **Coordinator Log (`txn.log`)**: Records `PREPARE`, `COMMITTED`, and `ABORTED` states.
- **Coordinator Queue (`queue.json`)**: Stores pending transactions.
- **KV Backend Log (`kv_backend_X.log`)**: Records `PREPARE`, `COMMIT`, `ABORT`, and `DISCARD` events for crash recovery and PostPrepare.

E. Client Protocol

The client supports CLI commands for `transfer`, `credit`, and `debit`. The flow is:

1) Submit the transaction to the Coordinator via `Txn`.
2) Wait for a commit notification via `ReceiveTxnStatus` or poll via `GetStatus`.
3) Acknowledge the transaction using `AckTxn`.

F. System Flow and Execution Paths

1) Normal Flow:

1) Client submits a transaction via `Txn`.
2) Coordinator enqueues and logs it.
3) `QueueWorker` sends `Prepare` RPCs to all KV servers.
4) If all servers reply `PREPARED`, Coordinator logs `TxnPrepared`.
5) Coordinator sends `Commit` RPCs, logs `COMMITTED`, and notifies the client.
6) Client sends `AckTxn` to complete the protocol.

2) Possible Alternate Flows and Outcomes:

- **Prepare Rejected**: If a backend rejects the `Prepare` (e.g., insufficient balance), Coordinator logs `ABORT` and notifies all backends.
- **Partial Commit**: If Coordinator crashes after some `Commit` messages, the recovery process will ensure `Commit` is re-driven for missing participants.
- **Client Crash**: If the client does not acknowledge, the system remains consistent and does not block.

G. Test Coverage

We designed an extensive test suite to validate functional correctness and recovery from failures. Tests include:

- Normal end-to-end flows for all transaction types.
- Coordinator crash cases: C1–C7 (run via `run_tests.sh`).
- Backend crash cases: S1–S6 (run via `run_tests_server.sh`).
- Manual polling, duplicate transactions, and ACK replays.

All tests confirm system guarantees of atomicity, consistency, and durability across varying concurrency, backend mappings, and crash events.

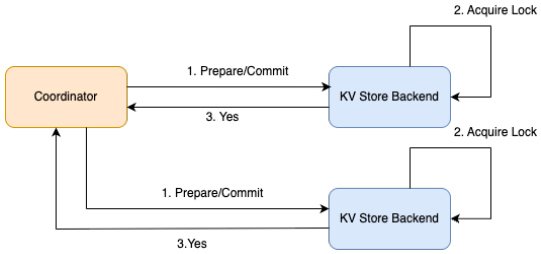

Fig. 1. Distributed Transaction Processing System using 2PC with Message Flow

H. Implementation Challenges

Implementing a recoverable 2PC protocol required solving several critical challenges. Developing a reliable termination protocol for unresolved transactions involved combining durable logging with Coordinator and peer queries while ensuring no transaction was left in an uncertain state. Idempotent handling of repeated `Commit`/`Abort` messages was essential to support retries during recovery. Coordinator and backend recovery flows had to be carefully engineered to reconstruct consistent state without conflict. Finally, writing automated test scripts that could simulate crash timing across protocol steps required precise control over process orchestration and synchronization.

IV. Evaluations

A. C1 – Normal Transaction Execution

This test evaluates the baseline correctness of the Two-Phase Commit (2PC) protocol under normal, failure-free conditions. The transaction `credit A 100` is initiated by the client and processed through the full 2PC flow.

Execution Flow:

- The Coordinator receives the transaction and issues `Prepare` messages to both backend servers.
- Both backends respond with `YES` (i.e., `PREPARED`) and log the `PREPARE` state.
- The Coordinator logs the `COMMIT` decision and sends `Commit` messages to the participants.
- Backend logs `COMMIT` and applies the update.
- The Coordinator sends the final `COMMITTED` status to the client.
- The client acknowledges the transaction with `AckTxn`.

Outcome: The transaction commits successfully with no failures or delays. This validates the standard success path of the 2PC protocol in our system and confirms that log persistence and protocol sequencing are functioning as expected.

B. C2 – Crash after Sending Prepare

This test evaluates the system's recovery behavior when the Coordinator crashes immediately after sending `Prepare` messages but before receiving all participant responses.

Execution Flow:

- The client initiates a `transfer A B 10` transaction.
- The Coordinator reaches the test hook at `PAUSE_AT_C2` and pauses after sending the `Prepare` messages to both backend servers.
- The Coordinator is forcefully terminated before collecting any `Prepared` responses.
- After restart, the Coordinator replays its logs, identifies the transaction in the `PENDING` state, and re-initiates the prepare phase.
- Backends receive either fresh or duplicate `Prepare` messages, respond accordingly, and the Coordinator proceeds to `COMMIT` upon receiving all affirmative responses.
- The transaction completes and the client receives a `COMMITTED` response.

Outcome: This test validates the Coordinator's ability to recover mid-prepare by replaying its logs and safely continuing the 2PC protocol. It demonstrates correctness of log persistence, idempotent messaging, and transaction consistency across failures.

C. C3 – Crash after Prepare ACKs Received but Before Commit

This test evaluates the Coordinator's ability to recover and commit a transaction after crashing at a critical point—after receiving all `Prepared` acknowledgments but before sending the `Commit`.

Execution Flow:

- A `transfer A B 20` transaction is initiated by the client.
- The Coordinator sends `Prepare` messages to the backends and receives affirmative `Prepared` responses.
- The test hook `PAUSE_AT_C3` halts the Coordinator immediately after recording the successful responses but before broadcasting the `Commit`.
- The Coordinator crashes and is restarted.
- Upon restart, it replays the log, detects the transaction as ready to commit, and proceeds with the `Commit` phase.
- Backends accept the `Commit` messages, apply the changes, and finalize the transaction.
- The client receives confirmation of a successful commit.

Outcome: This test validates correct transaction completion when the Coordinator crashes after gathering all votes but before initiating the commit. It confirms that the recovery mechanism ensures atomicity by relying on logged decisions rather than re-evaluating state.

D. C4 – Crash after TxnPrepared Log Written

This test examines the system's behavior when the Coordinator crashes after logging the `TxnPrepared` state but before sending any `Commit` messages to the backend servers.

Execution Flow:

- The client initiates a `transfer A B 25` transaction.
- The Coordinator completes the `Prepare` phase and logs the `TxnPrepared` state to disk.
- Before it can send out `Commit` messages, the Coordinator is halted via the test hook `PAUSE_AT_C4`.

- Upon restart, the Coordinator replays its transaction log and resumes the transaction from the `Prepared` state.
- It proceeds to send `Commit` messages to all participants.
- Backends complete the transaction and the client receives a confirmation of success.

Outcome: This test confirms the durability of the `TxnPrepared` log entry. Despite crashing before broadcasting the commit, the Coordinator is able to resume the transaction seamlessly after restart. This illustrates that once a commit decision is logged, recovery can safely complete the transaction without inconsistencies.

E. C5 – Crash after Some Commit Sent

This test examines the recovery behavior when the Coordinator crashes after partially completing the commit phase—i.e., after sending the `Commit` message to a subset of backend servers.

Execution Flow:

- A `transfer A B 10` transaction is submitted by the client.
- The Coordinator logs the `TxnPrepared` state and begins broadcasting `Commit` messages.
- It sends the `Commit` message to one backend but crashes before reaching the other.
- On restart, the Coordinator replays the log and resumes the `Commit` phase.
- The backend that previously did not receive the `Commit` now processes it upon retry.
- Both backends complete the commit, ensuring consistent transaction state.

Outcome: This test validates idempotent commit delivery and the system's ability to handle partial message dissemination. Upon recovery, the Coordinator correctly completes the commit process across all participants, ensuring atomicity despite mid-phase failure.

F. C6 – Crash after Full Commit but Before Client ACK

This test evaluates the system's robustness when the Coordinator crashes after all participants have successfully committed the transaction, but before the client receives the final acknowledgment.

Execution Flow:

- A `transfer A B 10` transaction is initiated by the client.
- The Coordinator completes the `Prepare` phase and all backends log the `Commit` after receiving the `Commit` message.
- Before the client is notified, the Coordinator pauses at the `PAUSE_AT_C6` hook and is terminated.
- Upon restart, the Coordinator replays its logs, identifies the transaction as fully committed, and re-sends the commit result to the client.
- The client acknowledges the transaction, completing the protocol.

Outcome: This test confirms durability of committed transactions. Even if the Coordinator crashes after the commit phase, it can accurately determine transaction state from logs and ensure client notification upon recovery, maintaining consistency and liveness.

G. C7 – Normal Commit with Delayed Client ACK

This test evaluates the system's ability to tolerate delays in client acknowledgment after a successful commit phase. Such delays may be caused by network latency, client-side issues, or deferred execution logic.

Execution Flow:

- A `transfer A B 20` transaction is submitted by the client.
- The Coordinator executes the full 2PC flow: sending `Prepare` to backends, receiving `Prepared` responses, logging `TxnPrepared`, and sending `Commit` to all participants.
- All backends apply the commit and confirm successful execution.
- The Coordinator returns the `COMMITTED` status to the client.
- The client delays sending `AckTxn` either intentionally or due to simulation logic.
- Eventually, the client sends the acknowledgment, allowing the Coordinator to perform final cleanup.

Outcome: This test confirms that client acknowledgment is not a requirement for the commit's durability or correctness. The system tolerates delayed ACKs and completes cleanup once the acknowledgment is received, demonstrating liveness and resilience to delayed client interactions.

H. S1 – Backend Crash Before Prepare

This test evaluates system behavior when a backend crashes before it receives any `Prepare` message from the Coordinator. The goal is to ensure that the system maintains safety and avoids unintended side effects in the absence of any transactional staging.

Execution Flow:

- A transaction is initiated by the client and passed to the Coordinator.
- One of the backend servers crashes before the Coordinator begins the `Prepare` phase.
- As a result, the crashed server has no staged logs or transactional state.
- Upon recovery and restart, the backend has no relevant entries in its log and thus no resolution or rollback is required.
- The Coordinator detects the timeout or failure and safely aborts the transaction.

Outcome: This test confirms that backend failures prior to transactional engagement do not introduce inconsistencies. The system handles the crash gracefully by not involving the server in any further steps, ensuring safety and aborting cleanly without requiring resolution during recovery.

I. S2 – Backend Crash after Prepare Written but Before ACK Sent

This test examines the behavior when a backend crashes after persisting the `PREPARE` log entry but before sending its `Prepared` acknowledgment to the Coordinator.

Execution Flow:

- The Coordinator begins a transaction and sends `Prepare` messages to all relevant backends.
- A backend receives the `Prepare`, writes the `PREPARE` log entry, but crashes before replying with `Prepared`.
- After recovery, the backend invokes `RecoverFromLog`, detects the incomplete transaction state, and runs `PostPrepare`.
- The backend queries the Coordinator or peers to determine the final transaction decision.
- Based on the query results, it applies the appropriate resolution: `Commit`, `Abort`, or `Discard`.

Outcome: This test confirms that even if a backend crashes mid-prepare, the system preserves correctness via persistent logging and resolution protocols. `PostPrepare` ensures that partial state is reconciled into a consistent outcome, maintaining safety and completeness.

J. S3 – Backend Crash after Prepare ACK, before Commit

This test explores the scenario where a backend acknowledges the `Prepare` message but crashes before receiving the `Commit` instruction from the Coordinator.

Execution Flow:

- The Coordinator initiates a transaction and receives `Prepared` responses from all participants.
- Before sending the `Commit` message to one of the backends, that server crashes.
- On restart, the backend invokes `RecoverFromLog` and detects it is in a `PREPARED` state without having committed.
- `PostPrepare` is triggered to query the Coordinator or peer backends to resolve the transaction outcome.
- If a `COMMIT` decision is found, the backend proceeds to apply and log the commit.

Outcome: This test validates correct handling of incomplete commit propagation. Persistent logging combined with `PostPrepare` ensures that all participants reach a consistent decision—even if they missed the original commit message—preserving atomicity of the 2PC protocol.

K. S4 – Backend Crash after Partial Commit Applied

This test evaluates system behavior when a backend crashes mid-way through applying the `Commit` phase—after logging the commit but before applying or finishing the full update.

Execution Flow:

- The Coordinator receives all `Prepared` responses and broadcasts `Commit` messages to all participants.
- A backend begins processing the `Commit`—possibly logging it or applying it partially—and then crashes unexpectedly.
- Upon recovery, the backend invokes `RecoverFromLog`, detects the presence of a `COMMIT` entry, and restores the key-value store to a consistent committed state.
- The recovery logic ensures that any partial state is completed or corrected based on the logged commit.

Outcome: This test confirms that committed transactions are idempotently re-applied on recovery. Persistent logs drive consistent reconstruction of the backend store, ensuring no data corruption even if the commit was interrupted.

L. S5 – Backend Crash after Commit Fully Done

This test evaluates the system's recovery behavior when a backend crashes after fully completing the commit phase and applying all changes to the store.

Execution Flow:

- The Coordinator completes the two-phase commit protocol and all backends receive and process the `Commit` message.
- A backend successfully logs the `COMMIT` entry and applies the corresponding transaction updates to its store.
- Afterward, the backend crashes.
- Upon restart, the backend invokes `RecoverFromLog` and replays the `COMMIT` entry to restore the post-commit state.
- Since the commit was already applied, the recovery phase confirms consistency and no further action is needed.

Outcome: This test demonstrates the durability of the commit operation. Once the commit is logged and applied, recovery ensures the store remains consistent and up to date, validating correct idempotent recovery behavior post-commit.

M. S6 – Backend Crash after Abort Written

This test validates the backend's recovery behavior after crashing mid-abort—i.e., after writing the `ABORT` log entry but before completing any post-abort cleanup.

Execution Flow:

- The Coordinator sends a `Prepare` to all backends, but a decision is made to abort the transaction due to a negative response or timeout.
- A backend receives the `Abort` instruction and logs the `ABORT` entry.
- It then crashes before completing further processing or cleanup.
- Upon restart, `RecoverFromLog` detects the `ABORT` state and ensures that no partial changes are applied.
- The backend transitions to a clean, safe post-abort state without impacting the data store.

Outcome: This test confirms that the abort logic is safely recoverable. Once the abort is logged, the backend can safely finalize or reinitialize its state on restart without risking partial updates or corruption.

We have comprehensively evaluated the correctness and fault tolerance behavior of our system through a suite of structured test cases (C1–C7, S1–S6), covering both normal

TABLE I
2PC FUNCTIONAL CORRECTNESS AND FAULT TOLERANCE SCENARIOS AND RECOVERY PATHS

Case ID	Scenario	Trigger Point	Recovery Path	Expected Outcome
C1	Normal Transaction (all ops)	Full 2PC flow	No recovery needed	All Prepare and Commit succeed; client receives COMMITTED
C2	GC crash before Prepare	GC crash after Txn received	QueueWorker restarts → Prepare phase reattempted	Transaction proceeds normally or ABORTED if timeout
C3	GC crash after some Prepare sent	Partial Prepare sent	On restart → Prepare phase restarted → safe idempotence	Commit or Abort based on Prepare success
C4	GC crash after all Prepare success, before TxnPrepared logged	After Prepare success	On restart → Prepare phase reattempted	Commit if safe, otherwise Abort
C5	GC crash after TxnPrepared logged, before Commit	After TxnPrepared	On restart → CommitPhase executed	All backends must Commit
C6	GC crash after partial Commit sent	Some Commit sent	On restart → CommitPhase retried	Commit must complete on all backends
C7	GC crash after Commit sent, before ACK to client	After Commit fully sent	On restart → AckTxn can be replayed by client	Coordinator must respond with COMMITTED
S1	Backend crash before Prepare	Before Prepare	On restart → nothing staged	Safe, no Prepare effect
S2	Backend crash after Prepare written but before ACK sent	During Prepare	On restart → RecoverFromLog → PostPrepare runs	PostPrepare pulls final decision (Commit/Abort/Discard)
S3	Backend crash after Prepare ACK, before Commit	During Commit phase	RecoverFromLog → PostPrepare triggers if no Commit seen	Commit must complete if decided
S4	Backend crash after partial Commit applied	Mid Commit	RecoverFromLog ensures store matches Commit	Store consistent with Commit decision
S5	Backend crash after Commit fully done	Post Commit	RecoverFromLog restores committed state	Store correct
S6	Backend crash after Abort written	Mid Abort	RecoverFromLog → prepares safe state	No effect on store

and failure scenarios across the Two-Phase Commit (2PC) protocol. Our automated testing framework allows us to configure various combinations of coordinator and backend crashes at different protocol phases, validating that all recovery paths (Prepare retry, Commit retry, PostPrepare handling, and safe DISCARD) are correctly handled.

To ensure robustness, we tested our system across multiple configurations—varying the number of bins from 1 to 8, and the number of backend servers from 1 to 4—thereby validating the correct routing, bin mapping, and resilience of the system under different data partitioning strategies. Across these configurations, every single flow was exhaustively tested to verify that the system guarantees safety and correctness even under concurrent coordinator and backend restarts. We explicitly validated the correctness of the Termination Protocol by verifying that backends correctly query both the coordinator and peer backends to resolve uncertain transaction outcomes.

All test cases (C1–C7, S1–S6) passed under multiple repeated runs, including randomized timing delays to simulate race conditions. We also tested recovery after mid-phase coordinator and backend crashes to ensure idempotence of Prepare and Commit phases and proper handling of late messages.

In future work, we aim to expand the testing suite with additional cases, including:

- Partial backend partition failures with asymmetric message visibility,
- Client reconnection and duplicate transaction submission scenarios,
- Concurrent submission of independent transactions to validate parallelism safety.

Additionally, after introducing controlled concurrency support into the system, we plan to perform detailed performance evaluation across different concurrency levels, backend counts, and transaction mixes, measuring throughput, latency, and recovery overhead.

V. LIMITATIONS AND FUTURE WORK

While the system successfully implements a fault-tolerant distributed transaction protocol using the Two-Phase Commit (2PC) protocol, several extended tests expose nuanced edge cases that highlight design assumptions, limitations, and potential areas for enhancement.

Crash Resilience Beyond Basic 2PC:

- **CX1 – Coordinator and Partial Backend Crash During Prepare:** If the GC and some servers crash during the prepare phase at random points, recovery depends on reattempted prepares via QueueWorker and resolution through PostPrepare. While recovery completes correctly, the process depends on log consistency and timely peer responses.
- **CX2 – Coordinator and Partial Backend Crash During Commit:** A crash during the commit phase requires

the Coordinator to reenter the commit phase on restart. Although global commit completes, it highlights the need for robust fencing mechanisms to prevent duplicate commits.

Network Partition Scenarios:

- **N1 – Partition During Prepare:** If the GC is partitioned from a subset of servers during prepare, timeouts correctly trigger an abort, maintaining consistency. However, the protocol incurs latency under high partition probability.
- **N2 – Partition During Commit:** When partitions occur after prepare but before commit, PostPrepare logic ensures transaction completion. Nevertheless, healing of the partition is required for forward progress.
- **N3 – Partition Between Backends:** In scenarios where backends cannot query each other (e.g., PostPrepare fails), the fallback resolution may result in a transaction being discarded (DISCARD) despite prior commit decisions being recorded elsewhere. Although safe, it is overly conservative and leads to potential loss of committed effects.

Client Fault Tolerance and Retry Logic:

- **CL1 – Client Crash Before ACK:** If a client crashes after submitting a transaction but before acknowledging it, it can safely retry using `GetStatus`, and the Coordinator replies with the correct transaction status.
- **CL2 – Client Crash After Seeing Commit:** If the client crashes after receiving a `COMMITTED` response but before sending `AckTxn`, the Coordinator retains the transaction in its in-memory map and correctly finalizes on receiving the retry.
- **CL3 – Restarted Client `GetStatus` Flow:** After a client restart, the Coordinator responds appropriately to repeated `GetStatus` queries, ensuring durability and user-visible consistency.

Timing and Race Conditions:

- **T1 – Late Prepare ACK:** Late-arriving Prepare responses after the Coordinator has already timed out and aborted are handled correctly by the backend through PostPrepare.
- **T2 – Late Commit ACK After GC Crash:** Delayed commit acknowledgments arriving after Coordinator restart are safely handled due to idempotent commit logic.
- **T3 – DISCARD Despite Commit Observed:** Conflicting PostPrepare outcomes between backends are possible. A backend may consider discarding if peer querying fails, but the protocol ensures that if even one peer confirms a commit, a DISCARD is not issued.

Future Work:

- **Concurrency Support:** The current system handles one transaction at a time in a globally serialized fashion. Future iterations will support concurrent transactions by tracking key-level conflicts, enabling non-conflicting transactions to proceed in parallel. This would significantly improve throughput and reduce contention in high-load environments.
- **Smarter Failure Prediction and Backoff:** Introducing adaptive backoff and heartbeat-driven liveness checks may help mitigate latency in network partitions and minimize unnecessary aborts.

VI. Conclusion

In this project, we designed and implemented a robust distributed transaction processing system using the Two-Phase Commit (2PC) protocol. Our architecture, composed of a Go-based Coordinator and Rust-based backend KV servers, ensures atomicity and durability across multiple storage nodes despite the presence of failures. By incorporating persistent logging, PostPrepare resolution, and a Termination Protocol, our system recovers cleanly from a wide range of crash and network fault scenarios, including mid-phase coordinator failures and uncertain backend states.

The evaluation through automated test cases (C1–C7, S1–S6) confirms the correctness of our implementation under normal and adversarial conditions. Furthermore, our system maintains strict transactional semantics while preserving data consistency across distributed bins. Our logging and recovery mechanisms ensure that no transaction is left in an indeterminate state, and all components can safely rejoin the system after recovery without violating protocol integrity.

Although our current implementation serializes all transactions for correctness, we have laid a solid foundation for scaling toward concurrent transaction processing and more advanced failure-handling strategies. Future work will focus on extending concurrency support, optimizing recovery latency, and minimizing false aborts under partitions or transient failures.