JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

DISTRIBUTED COMPUTING AND APPLICATIONS

ICS 2403

FAULT-TOLERANT DISTRIBUTED TRANSACTIONS

GROUP 3

NAME	**REGISTRATION NUMBER**
George Okuthe	*ENE221-0114/2021*
Catherine Atieno	*ENE221-0143/2021*
Debra Omollo	*ENE221-0133/2021*
Nicole Mbodze	*ENE221-0134/2021*
Fortune Otieno	*ENE221-0125/2021*
Fiona Mitchelle	*ENE221-0113/2021*

INTRODUCTION

This project is a small-scale distributed transaction system implemented in Python to illustrate Atomic distributed transactions coordinated across multiple nodes, concurrency control using locks (2PL-style), and failure handling and recovery with simple write-ahead logging and restart behavior. The system is a 4-node architecture consisting of 1 coordinator and 3 data nodes.

High-Level Idea

We modelled a bank transfer as a transaction with a debit account `A` on one node and a credit account `B` on another node. The system comprises of a **Coordinator**, the central transaction manager that:

- Receives transaction requests from clients
- Orchestrates **the two-phase commit (2PC) protocol** across data nodes
- Makes the final commit/abort decision to ensure atomicity
- Does NOT store account data (only coordinates transactions)

Data Nodes labeled as N1, N2, N3 that each store a subset of accounts and enforce locking and local checks. **Clients** which are multiple instances that initiate transactions (e.g., bank customers, ATM users, or banking applications requesting transfers). Our system is designed to implement, and demonstrate a small-scale distributed application that integrates;

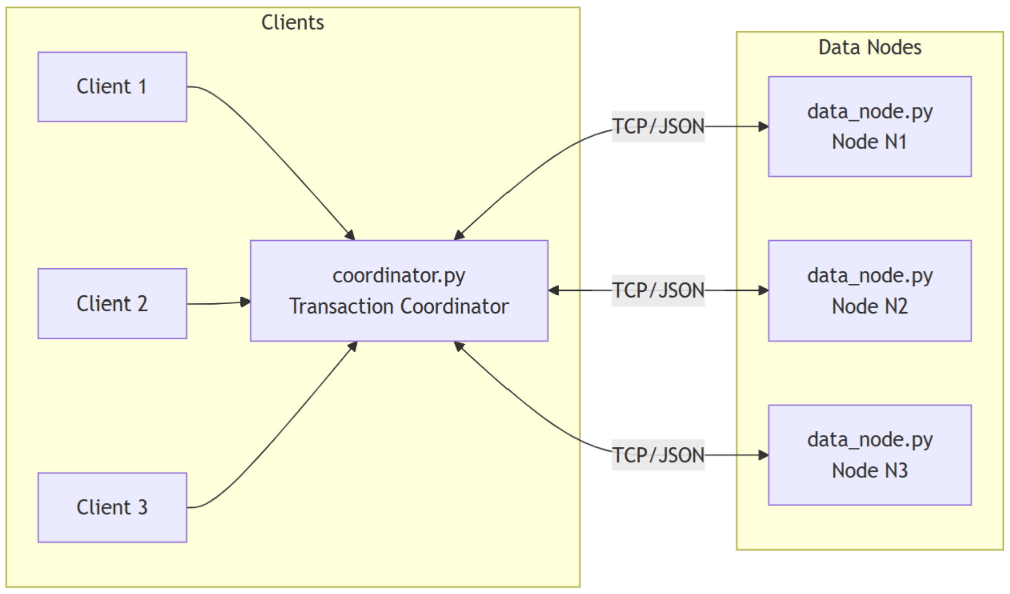
1. **Atomicity**
   This is achieved by ensuring that either both debit and credit happen (**all nodes commit**) or nothing happens at all (**all nodes abort**).

2. **Concurrency control**
   Our implementation relies on nodes that use per-account locks to serialize conflicting updates (basic 2-phase locking).

3. **Failure handling**
   Each Node has a write-ahead log and a state file. If a data node crashes and restarts, it can still resume from its last committed state, and in-flight transactions are safely aborted from the coordinator's point of view.

System Architecture Diagram

The following diagram shows the high-level system architecture and how components interact:



The architecture diagram shows:
Clients: Multiple client processes (client.py) can run concurrently. The diagram shows 3 clients(Client 1, Client 2, Client 3). Each client:

- Represents an entity that initiates transactions (e.g., a bank customer, ATM user, or banking application)
- Initiates transactions (bank transfers between accounts) by sending `TRANSFER` requests to the coordinator over TCP/JSON
- Runs independently in separate terminal windows or processes (CLI)
- Can send concurrent requests to test the system's concurrency control

Coordinator: The coordinator.py process acts as the central transaction manager. It:

- Receives transaction requests from clients
- Orchestrates the two-phase commit protocol across data nodes
- Makes the final commit/abort decision to ensure atomicity
- Does NOT store account data itself (only coordinates)

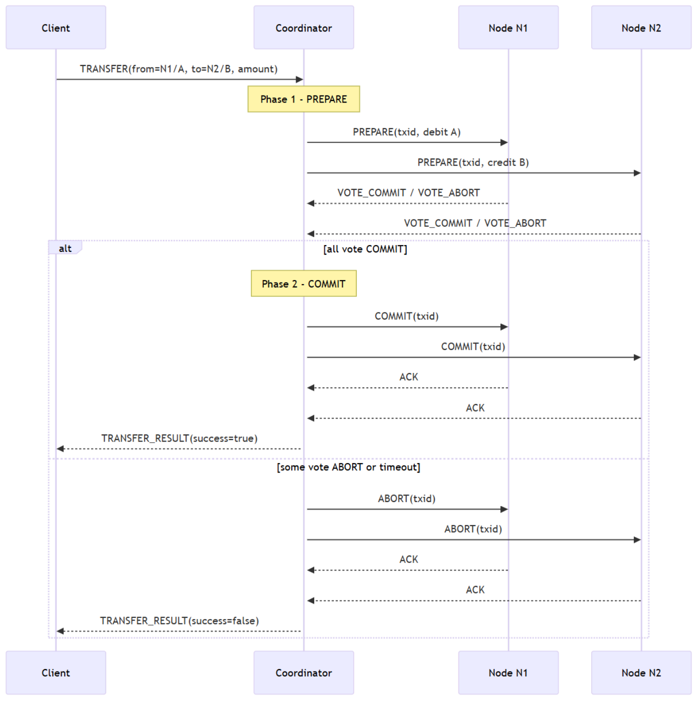
Data Nodes: Exactly 3 independent data_node.py processes, labeled as: N1(runs on port 6001, N2 (runs on port 6002), and N3 (runs on port 6003). Each data node:

- Stores actual account balances (partitioned across nodes)
- Process sub-transactions (debits/credits) for accounts they manage
- Uses locks for concurrency control
- Persists state to disk for crash recovery

All **communication** uses **TCP sockets** with JSON-encoded messages. The coordinator communicates bidirectionally with each data node to execute the 2PC protocol.

Transaction Flow Sequence Diagram (2PC Protocol)

The following sequence diagram illustrates exactly what happens during a single distributed transaction:



This sequence diagram shows:
Client Request: A client sends a `TRANSFER` request to the coordinator specifying:

- The Source: `N1/A` (account A on node N1),
- Destination: `N2/B` (account B on node N2),
- Amount to transfer.

Phase 1 - PREPARE: This is the voting phase where the coordinator generates a unique transaction ID (`txid`) then sends `PREPARE` messages to both involved nodes (N1 and N2). Each node:

- acquires locks on the affected account(s)
- validates the operation (e.g., checks if debit would cause negative balance)
- logs the prepare intent
- replies with `VOTE_COMMIT` (if feasible) or `VOTE_ABORT` (if not)

Phase 2 - COMMIT or ABORT: This is the Decision phase where:

- **If all nodes vote COMMIT**:
    1. The Coordinator sends `COMMIT` messages to all involved nodes.
    2. The nodes apply the changes permanently (update balances, persist to disk)
    3. Nodes send `ACK` back to the coordinator
    4. Coordinator returns `success=true` to the client.
- **If any node votes ABORT or times out**:
    1. The Coordinator sends `ABORT` messages to all involved nodes.
    2. The nodes release locks and discard the prepared changes
    3. Nodes send `ACK` back to the coordinator
    4. Coordinator returns `success=false` to the client.

Atomicity Guarantee: The transaction is all-or-nothing, either both debit (N1) and credit (N2) happen together or neither happens (both abort). This ensures the system never enters an inconsistent state where money disappears or appears from nowhere.

Concurrency Control Strategy

Each node uses a dictionary of balances, e.g., {"A": 100, "B": 50, ...} and a dictionary of locks, one per account. During the PREPARE phase, the nodes lock all involved accounts in a sorted order and simulate updates on temporary balances:

- On success, the node:
    1. Logs a `prepare_ok` record.
    2. Sends `VOTE_COMMIT`.
- On failure (e.g., negative balance), the node:
    1. Sends `VOTE_ABORT`.
    2. Locks are released at the end of PREPARE.

On COMMIT, for each operation, the node:

1. Acquires the account lock.
2. Logs the delta in the write-ahead log.
3. Updates the in-memory balance.
4. Writes updated state to the state file.
5. Releases the lock.

This corresponds to a basic 2PL-like mechanism: locks are taken during critical sections to prevent concurrent conflicting accesses, ensuring serializable behavior for this simple workload.

Failure Handling and Recovery

This implementation demonstrates failure scenarios and recovery behavior, at a conceptual level. Some of the scenarios that are likely to cause failure are highlighted below:

1. **Data-node crash**

Every node maintains a state file (showing last committed account balances) and a log file (with append-only records of transaction events and updates). If a node process is killed, The In-memory state for in-flight transactions is lost but the persisted state remains consistent because:

- Data is written before or during commit.
- No partial writes across transactions are exposed.

On restart, the node reloads the state file (last consistent snapshot). Any transaction that was in PREPARE but not committed is effectively aborted (coordinator will time out and abort).

2. **Coordinator sees node failure**

If a node does not respond to PREPARE or COMMIT the coordinator interprets this as a vote to abort and:

- Sends `ABORT` to all other involved nodes.
- Reports `success=false` to the client.

3. **Coordinator crash and recovery**

The coordinator maintains a transaction log file (`data/coordinator_tx_log.jsonl`) that records:

- Transaction start (`START`)
- Prepare phase (`PREPARE`)
- Commit/abort decisions (`COMMIT`, `ABORT`)
- Transaction completion (`COMPLETE`)

On coordinator restart, the coordinator scans the log for incomplete transactions (those that reached `PREPARE` or `COMMIT` but never reached `COMPLETE` or `ABORT`). For each incomplete transaction, the coordinator sends `ABORT` messages to all involved nodes, this ensures nodes never remain in an uncertain state (they only commit when explicitly told to).

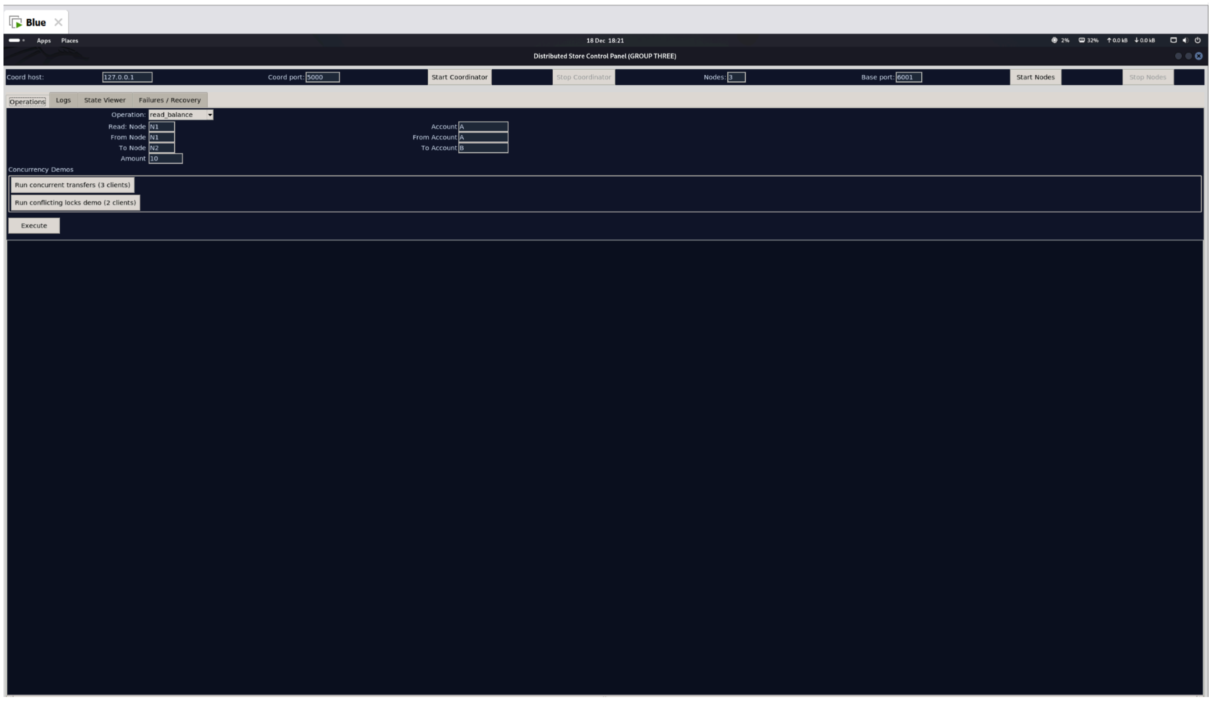
This technique for failure handling and recovery works because:

- Nodes never commit without an explicit `COMMIT` message from the coordinator.
- If the coordinator crashes after sending `PREPARE` but before `COMMIT`, nodes remain in a prepared state.
- On recovery, the coordinator aborts these incomplete transactions, ensuring consistency.
- The system resumes normal operation after recovery completes

Key Observations and Analysis

Using this architecture we were able to implement a small-scale distributed transaction system meeting all the requirements, i.e, an application that integrates transactions, concurrency control, and fault tolerance. Below are some of the screenshots and logs captured during testing:

1. **The Graphical User Interface**
   To streamline system interaction, we developed a centralized GUI to replace the manual CLI workflow. Previously, managing concurrent transactions via the CLI was error-prone, as it required synchronized manual inputs across multiple terminal instances. The new interface eliminates these bottlenecks by allowing users to trigger orchestrated, multi-client requests from a single control point, ensuring precise timing for concurrency testing.

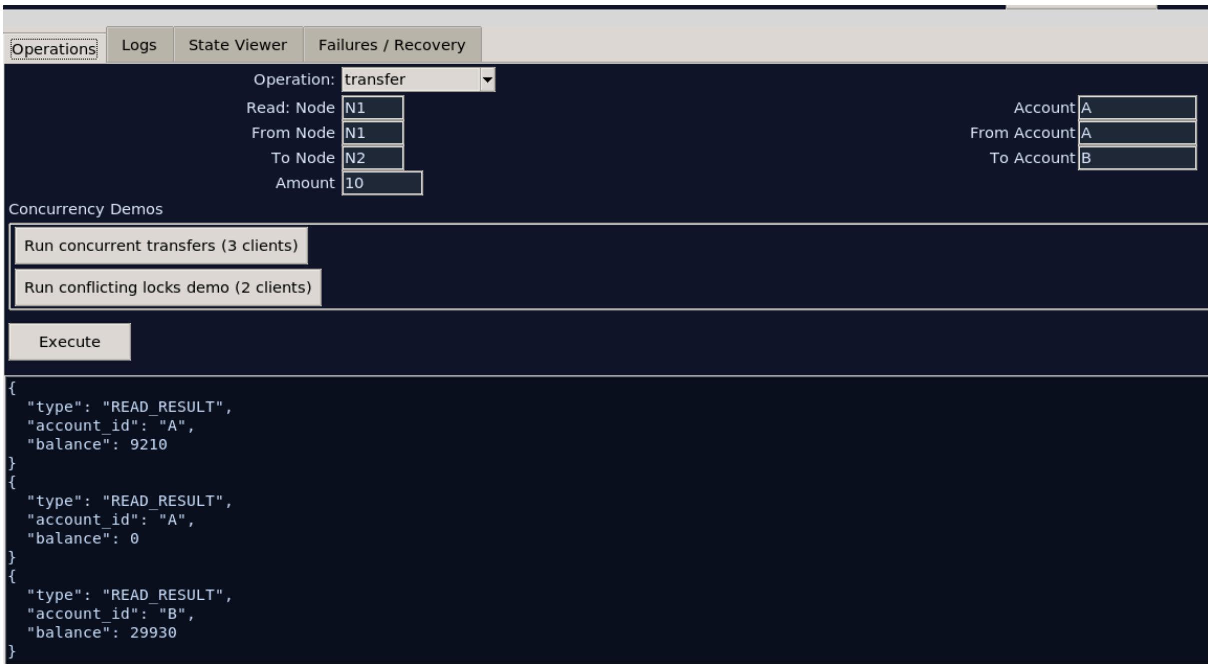


Distributed Store Control Panel (GROUP THREE)

Coord host: 127.0.0.1
Coord port: 5000
Nodes: 3
Base port: 6001

Operation: read_balance
Read: Node N1
From Node N1
To Node N2
Amount 10
Account A
From Account A
To Account B

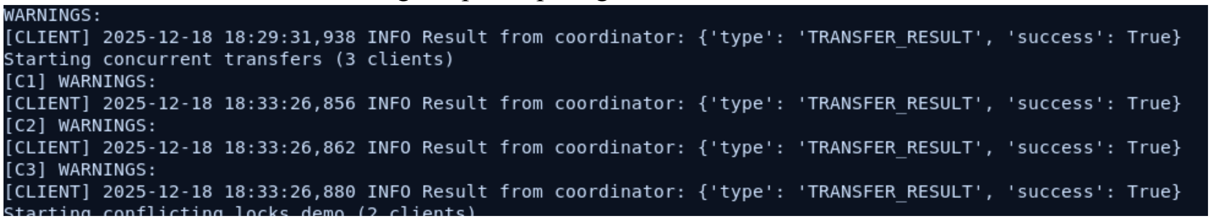
2. **Reading balances**
   The system provides a streamlined interface for querying account balances. To retrieve a balance, simply select your preferred **Read Node** and enter the **Account name**. This feature allows users to inspect the balance of a specific account (**Account A**) from the perspective of an individual node (**Node N1**), which is essential for identifying potential synchronization delays or replication lags.


Operations
Logs
State Viewer
Failures / Recovery
Concurrency Demos
Run concurrent transfers (3 clients)
Run conflicting locks demo (2 clients)
Execute

```
Operation: transfer
Read: Node N1                         Account A
From Node N1                     From Account A
To Node N2                         To Account B
Amount 10
```

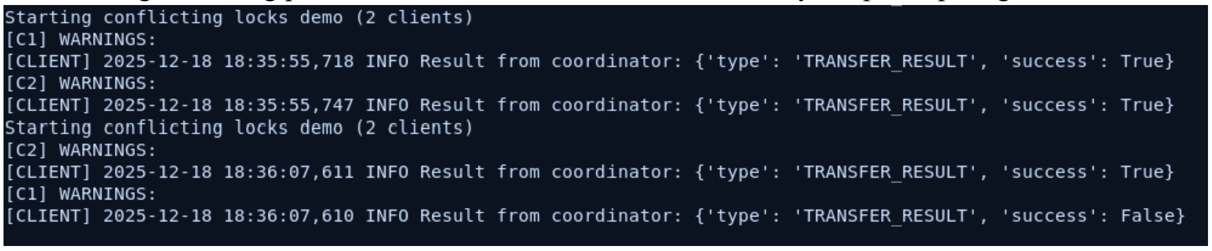
```
{
  "type": "READ_RESULT",
  "account_id": "A",
  "balance": 9210
}
{
  "type": "READ_RESULT",
  "account_id": "A",
  "balance": 0
}
{
  "type": "READ_RESULT",
  "account_id": "B",
  "balance": 29930
}
```

3. **Concurrent client operations(transactions) and concurrency control**
   The system supports multi-client concurrency, managed via a validation-based control mechanism. While multiple transactions can be initiated simultaneously, the system enforces strict serializability; a transaction is only finalized if the coordinator detects no resource contention or state conflicts among the participating nodes.

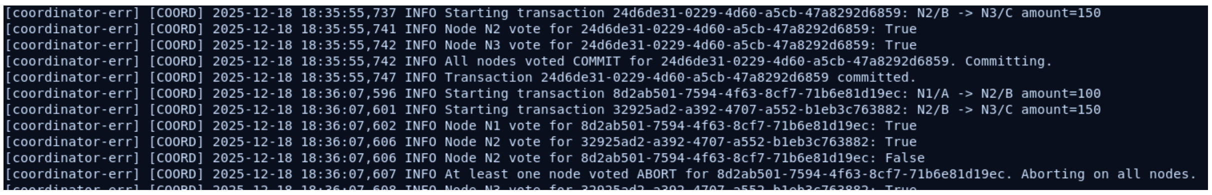


```
WARNINGS:
[CLIENT] 2025-12-18 18:29:31,938 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
Starting concurrent transfers (3 clients)
[C1] WARNINGS:
[CLIENT] 2025-12-18 18:33:26,856 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
[C2] WARNINGS:
[CLIENT] 2025-12-18 18:33:26,862 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
[C3] WARNINGS:
[CLIENT] 2025-12-18 18:33:26,880 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
Starting conflicting locks demo (2 clients)
```

4. **Transaction commit and Rollback**
   During concurrent operations, the Coordinator performs a validation check to identify potential resource contention. If no conflicts are detected, the Coordinator orchestrates a multi-node commit to finalize the transaction. However, if contention arises, a rollback is triggered to preserve data integrity. The following figures illustrate this workflow: the first displays the user-facing client interface, while the second provides a detailed execution log documenting the voting phase and the eventual consensus reached by the participating nodes.

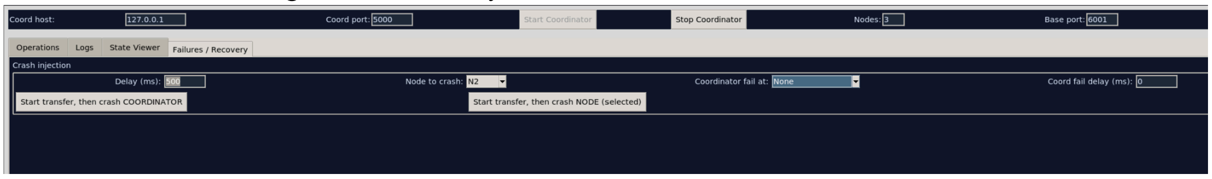


```
Starting conflicting locks demo (2 clients)
[C1] WARNINGS:
[CLIENT] 2025-12-18 18:35:55,718 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
[C2] WARNINGS:
[CLIENT] 2025-12-18 18:35:55,747 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
Starting conflicting locks demo (2 clients)
[C2] WARNINGS:
[CLIENT] 2025-12-18 18:36:07,611 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': True}
[C1] WARNINGS:
[CLIENT] 2025-12-18 18:36:07,610 INFO Result from coordinator: {'type': 'TRANSFER_RESULT', 'success': False}
```



```
[coordinator-err] [COORD] 2025-12-18 18:35:55,737 INFO Starting transaction 24d6de31-0229-4d60-a5cb-47a8292d6859: N2/B -> N3/C amount=150
[coordinator-err] [COORD] 2025-12-18 18:35:55,741 INFO Node N2 vote for 24d6de31-0229-4d60-a5cb-47a8292d6859: True
[coordinator-err] [COORD] 2025-12-18 18:35:55,742 INFO Node N3 vote for 24d6de31-0229-4d60-a5cb-47a8292d6859: True
[coordinator-err] [COORD] 2025-12-18 18:35:55,742 INFO All nodes voted COMMIT for 24d6de31-0229-4d60-a5cb-47a8292d6859. Committing.
[coordinator-err] [COORD] 2025-12-18 18:35:55,747 INFO Transaction 24d6de31-0229-4d60-a5cb-47a8292d6859 committed.
[coordinator-err] [COORD] 2025-12-18 18:36:07,596 INFO Starting transaction 8d2ab501-7594-4f63-8cf7-71b6e81d19ec: N1/A -> N2/B amount=100
[coordinator-err] [COORD] 2025-12-18 18:36:07,601 INFO Starting transaction 32925ad2-a392-4707-a552-b1eb3c763882: N2/B -> N3/C amount=150
[coordinator-err] [COORD] 2025-12-18 18:36:07,602 INFO Node N1 vote for 8d2ab501-7594-4f63-8cf7-71b6e81d19ec: True
[coordinator-err] [COORD] 2025-12-18 18:36:07,606 INFO Node N2 vote for 32925ad2-a392-4707-a552-b1eb3c763882: True
[coordinator-err] [COORD] 2025-12-18 18:36:07,606 INFO Node N2 vote for 8d2ab501-7594-4f63-8cf7-71b6e81d19ec: False
[coordinator-err] [COORD] 2025-12-18 18:36:07,607 INFO At least one node voted ABORT for 8d2ab501-7594-4f63-8cf7-71b6e81d19ec. Aborting on all nodes.
[coordinator-err] [COORD] 2025-12-18 18:36:07,608 INFO Node N3 vote for 32925ad2-a392-4707-a552-b1eb3c763882: True
```

5. **Failure Handling and Recovery**
   The system includes a **Fault Injection** framework designed to simulate component failures across various stages of the transaction lifecycle—specifically during the **Prepare phase**, or immediately before and after the **Commit**. To maintain high availability, the architecture implements a self-healing protocol: upon reboot, a failed Node recovers its last consistent state from persistent storage. Simultaneously, the Coordinator audits the transaction logs to resolve any 'in-flight' operations, either finalizing them from the last checkpoint or executing a full rollback to ensure global consistency.

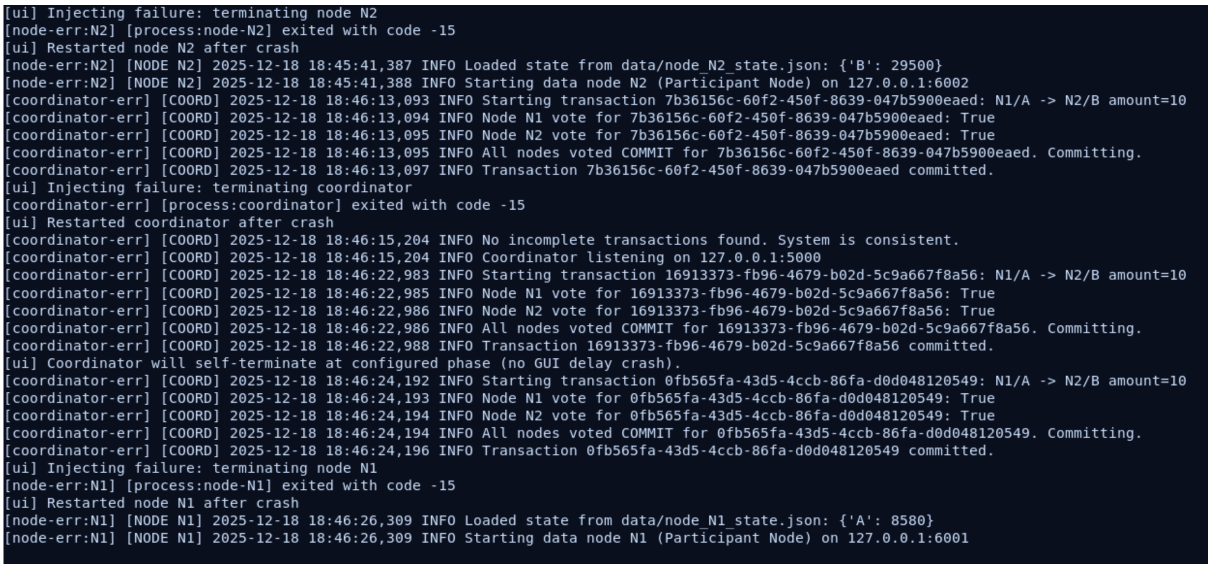


Coord host: 127.0.0.1 Coord port: 5000 Nodes: 3 Base port: 6001

Crash injection

Delay (ms): 500 Node to crash: N2 Coordinator fail at: None Coord fail delay (ms): 0

Start Coordinator
Stop Coordinator
Operations
Logs
State Viewer
Failures / Recovery
Start transfer, then crash COORDINATOR
Start transfer, then crash NODE (selected)



```
[ui] Injecting failure: terminating node N2
[node-err:N2] [process:node-N2] exited with code -15
[ui] Restarted node N2 after crash
[node-err:N2] [NODE N2] 2025-12-18 18:45:41,387 INFO Loaded state from data/node_N2_state.json: {'B': 29500}
[node-err:N2] [NODE N2] 2025-12-18 18:45:41,388 INFO Starting data node N2 (Participant Node) on 127.0.0.1:6002
[coordinator-err] [COORD] 2025-12-18 18:46:13,093 INFO Starting transaction 7b36156c-60f2-450f-8639-047b5900eaed: N1/A -> N2/B amount=10
[coordinator-err] [COORD] 2025-12-18 18:46:13,094 INFO Node N1 vote for 7b36156c-60f2-450f-8639-047b5900eaed: True
[coordinator-err] [COORD] 2025-12-18 18:46:13,095 INFO Node N2 vote for 7b36156c-60f2-450f-8639-047b5900eaed: True
[coordinator-err] [COORD] 2025-12-18 18:46:13,095 INFO All nodes voted COMMIT for 7b36156c-60f2-450f-8639-047b5900eaed. Committing.
[coordinator-err] [COORD] 2025-12-18 18:46:13,097 INFO Transaction 7b36156c-60f2-450f-8639-047b5900eaed committed.
[ui] Injecting failure: terminating coordinator
[coordinator-err] [process:coordinator] exited with code -15
[ui] Restarted coordinator after crash
[coordinator-err] [COORD] 2025-12-18 18:46:15,204 INFO No incomplete transactions found. System is consistent.
[coordinator-err] [COORD] 2025-12-18 18:46:15,204 INFO Coordinator listening on 127.0.0.1:5000
[coordinator-err] [COORD] 2025-12-18 18:46:22,983 INFO Starting transaction 16913373-fb96-4679-b02d-5c9a667f8a56: N1/A -> N2/B amount=10
[coordinator-err] [COORD] 2025-12-18 18:46:22,985 INFO Node N1 vote for 16913373-fb96-4679-b02d-5c9a667f8a56: True
[coordinator-err] [COORD] 2025-12-18 18:46:22,986 INFO Node N2 vote for 16913373-fb96-4679-b02d-5c9a667f8a56: True
[coordinator-err] [COORD] 2025-12-18 18:46:22,986 INFO All nodes voted COMMIT for 16913373-fb96-4679-b02d-5c9a667f8a56. Committing.
[coordinator-err] [COORD] 2025-12-18 18:46:22,988 INFO Transaction 16913373-fb96-4679-b02d-5c9a667f8a56 committed.
[ui] Coordinator will self-terminate at configured phase (no GUI delay crash).
[coordinator-err] [COORD] 2025-12-18 18:46:24,192 INFO Starting transaction 0fb565fa-43d5-4ccb-86fa-d0d048120549: N1/A -> N2/B amount=10
[coordinator-err] [COORD] 2025-12-18 18:46:24,193 INFO Node N1 vote for 0fb565fa-43d5-4ccb-86fa-d0d048120549: True
[coordinator-err] [COORD] 2025-12-18 18:46:24,194 INFO Node N2 vote for 0fb565fa-43d5-4ccb-86fa-d0d048120549: True
[coordinator-err] [COORD] 2025-12-18 18:46:24,194 INFO All nodes voted COMMIT for 0fb565fa-43d5-4ccb-86fa-d0d048120549. Committing.
[coordinator-err] [COORD] 2025-12-18 18:46:24,196 INFO Transaction 0fb565fa-43d5-4ccb-86fa-d0d048120549 committed.
[ui] Injecting failure: terminating node N1
[node-err:N1] [process:node-N1] exited with code -15
[ui] Restarted node N1 after crash
[node-err:N1] [NODE N1] 2025-12-18 18:46:26,309 INFO Loaded state from data/node_N1_state.json: {'A': 8580}
[node-err:N1] [NODE N1] 2025-12-18 18:46:26,309 INFO Starting data node N1 (Participant Node) on 127.0.0.1:6001
```

Conclusion

In conclusion, this project successfully demonstrates the architectural requirements for maintaining ACID properties in a distributed environment. By implementing a robust Two-Phase Commit (2PC) protocol alongside strict Two-Phase Locking (2PL), the system ensures that multi-node transactions remain atomic and isolated, even under significant resource contention. Furthermore, the integration of write-ahead logging and coordinator-led reconciliation proves that data integrity can be preserved through arbitrary component failures. Ultimately, this implementation serves as a comprehensive model for how orchestration and persistence layers must interact to achieve global consistency across a distributed network.