**Implementing and Evaluating a Fault-Tolerant Two-Phase Commit Protocol in Distributed Systems**

**Manoj Kumar Nallamala**

**MAV ID: 1002075929**

**Index:**

1. **Abstract**
   * Summary of the project's objectives, methodology, and key findings.
2. **Introduction**
   * Background on Two-Phase Commit Protocol and the significance of fault tolerance in distributed systems.
3. **Project Overview**
   * Detailed description of the project scope and objectives.
4. **System Architecture and Implementation**
   * Overview of the system architecture including the Transaction Coordinator and participant nodes.
   * Implementation details of the protocol.
5. **Part 1: Coordinator Failure Before 'Prepare' Message**
   * Implementation strategy and testing results.
6. **Part 2: Transaction Abortion on Missing 'Yes' Responses**
   * Implementation strategy and testing results.
7. **Part 3: Coordinator Crash After Partial 'Commit' Messages**
   * Implementation strategy and testing results.
8. **Part 4: Participant Node Failure Post 'Yes' Reply**
   * Implementation strategy and testing results.
9. **Failure Simulation and Handling Techniques**
   * Methods used for simulating failures and handling them in the system.
10. **Results and Analysis**
    * Comprehensive analysis of test outcomes and system behavior under different failure scenarios.

**Abstract**

In this report, we present the implementation and evaluation of a fault-tolerant Two-Phase Commit (2PC) protocol in a distributed system. The protocol's resilience is tested under various failure scenarios, including Transaction Coordinator (TC) and participant node crashes. Our implementation emulates a distributed environment with one TC and two participant nodes, focusing on the protocol's response to controlled and random failures. The study highlights the challenges and intricacies involved in ensuring data consistency and system reliability in distributed transactions, especially under failure conditions. Our findings provide insights into the behavior of the 2PC protocol in adverse situations, contributing to the understanding of fault tolerance in distributed systems.

**Introduction**

**Background**

The Two-Phase Commit (2PC) protocol is a cornerstone in distributed database systems, ensuring all participants in a distributed transaction either commit or abort in a coordinated manner. This protocol is crucial for maintaining data consistency across a distributed system. However, its effectiveness can be significantly undermined by the inherent risks of node or coordinator failures. In such a context, understanding and improving the fault tolerance of the 2PC protocol is paramount.

**Importance of Fault Tolerance**

Fault tolerance is a critical aspect of distributed systems, especially in transaction management. A fault-tolerant system is designed to continue its operation, possibly at a reduced level, rather than failing completely, when some part of the system fails. In the realm of distributed transactions, this means ensuring that all participating nodes reach a consensus on either committing or aborting a transaction, even in the face of failures.

**Project Scope and Objectives**

This project focuses on implementing and evaluating a fault-tolerant 2PC protocol. The implementation involves one Transaction Coordinator (TC) and two participant nodes. Our objectives include:

* Developing a system that can handle various failure scenarios, such as TC failure before sending 'prepare' messages and node failures after responding 'yes'.
* Analyzing how the system recovers from these failures and maintains transaction consistency.
* Gaining insights into the challenges and best practices in implementing fault-tolerant systems in a distributed environment.

**Project Overview**

Project Scope

This project aims to implement a fault-tolerant Two-Phase Commit (2PC) protocol in a distributed system environment. The primary focus is on ensuring the system's ability to handle various scenarios of node and Transaction Coordinator (TC) failures. The project simulates different failure conditions to study the protocol's response and its ability to maintain transaction consistency and system reliability.

Objectives

The key objectives of this project include:

1. **Developing a Robust 2PC Protocol**: Create an implementation of the 2PC protocol that can manage transactions across multiple nodes in a distributed system.
2. **Handling Failures**: Implement mechanisms to handle failures at different stages of the transaction process, including TC failures before sending 'prepare' messages and participant node failures after responding 'yes'.
3. **Ensuring Consistency**: Ensure that the system maintains transactional consistency, even in the event of node or TC failures.
4. **Studying System Behavior**: Analyze how the system behaves under different failure scenarios, focusing on its ability to recover and maintain consistent transaction states.
5. **Learning and Documentation**: Document the challenges, solutions, and learnings encountered during the implementation, providing a reference for future developments in this area.

**System Architecture and Implementation**

System Architecture

The architecture of the system comprises the following components:

1. **Transaction Coordinator (TC)**: Acts as the central authority that initiates transactions, sends 'prepare' messages to participant nodes, and decides whether to commit or abort the transaction based on participant responses.
2. **Participant Nodes**: Nodes that participate in the transaction, responding to the TC's 'prepare' request with a 'yes' or 'no', and acting on the TC's final decision to commit or abort.

Implementation Details

1. **Transaction Initiation**: Transactions are initiated by the TC, which sends a 'start transaction' message to participant nodes.
2. **Prepare Phase**: The TC sends a 'prepare' message to all participant nodes. In case of a failure simulation, this step can be altered to test system behavior under failure conditions.
3. **Voting by Participants**: Participant nodes vote 'yes' or 'no' to the prepare request, based on their state and any simulated failure conditions.
4. **Commit or Abort Decision**: Based on the responses from participant nodes, the TC decides to commit or abort the transaction. If all nodes vote 'yes', the TC proceeds to commit; otherwise, it aborts.
5. **Failure Handling Mechanisms**: The system includes mechanisms to handle TC failures before sending 'prepare' messages, and participant node failures after responding 'yes'.
6. **Recovery and Consistency**: The implementation ensures that, upon recovery from a failure, the system checks the last known state and takes appropriate actions to maintain consistency.

**Part 1: Handling Coordinator Failure Before 'Prepare' Message**

5.1 Implementation Strategy

**Scenario Description**

In this part of the project, we address the scenario where the Transaction Coordinator (TC) fails before it can send the 'prepare' message to participant nodes. This situation is critical as it tests the system's ability to handle unexpected disruptions during the initial phase of the transaction process.

**Approach**

1. **Simulated Failure**: We introduce a controlled failure in the TC by deliberately preventing it from sending the 'prepare' message after initiating a transaction. This is achieved by incorporating a conditional delay or halt in the TC's execution flow before the 'prepare' message broadcast.
2. **Participant Node Response**: Participant nodes, upon not receiving the 'prepare' message within a predefined timeout, are programmed to assume a failure in the TC and move to abort the transaction.
3. **TC Recovery and Subsequent Actions**: After the simulated failure period, the TC is brought back online. The TC then attempts to send the 'prepare' message. However, participant nodes, having already decided to abort due to the timeout, respond with a 'no' to these late 'prepare' requests.
4. **Logging and Time-out Mechanism**: Both the TC and participant nodes maintain logs to record the transaction states and timeouts, which are crucial for understanding the system behavior during and after the failure.

5.2 Testing and Results

**Test Setup**

* **Failure Simulation**: We simulated the TC failure by introducing a temporary halt in the TC's operation before sending out 'prepare' messages.
* **Participant Node Monitoring**: Participant nodes were monitored to ensure they correctly identify the timeout and transition to the abort state.

**Results**

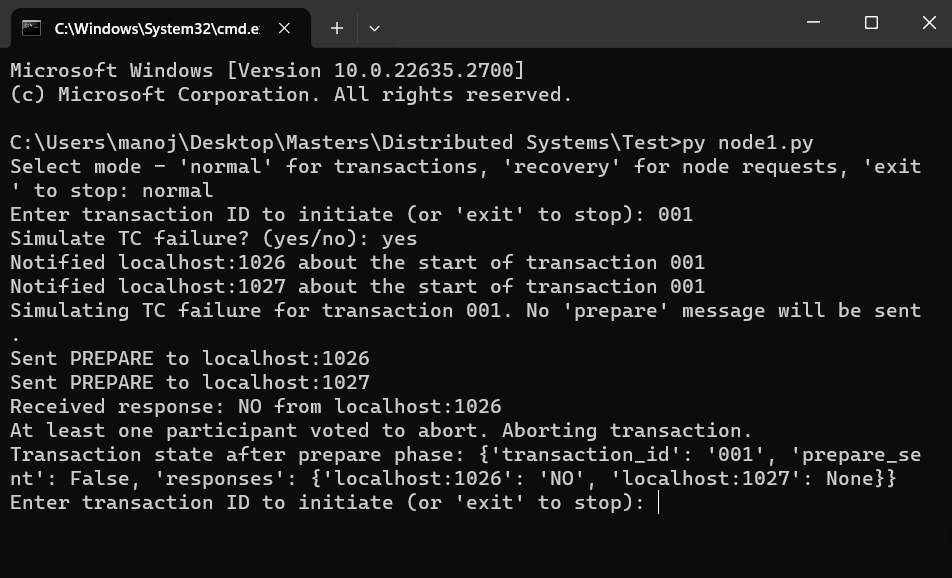
* **Participant Node Behavior**: As expected, upon not receiving the 'prepare' message within the timeout period, all participant nodes transitioned to an abort state.
* **TC Recovery**: When the TC recovered and attempted to send the 'prepare' message, it received 'no' responses from the participant nodes, as they had already moved to abort the transaction.
* **System Consistency**: Despite the failure of the TC, the system maintained consistency, with all nodes agreeing on aborting the transaction.

**Observations**

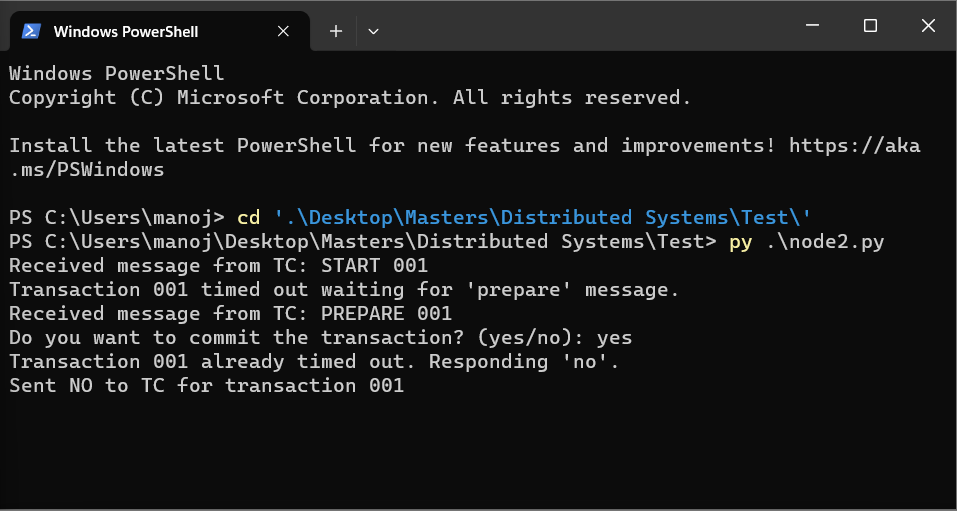
* **Importance of Timeouts**: This test highlighted the importance of having well-defined timeouts in a distributed system to handle failures effectively.
* **Recovery and State Management**: The ability of the TC to recover and attempt to proceed with the transaction showcases the resilience of the system, although the outcome was an abort due to the participant nodes' timeout logic.

**Conclusion**

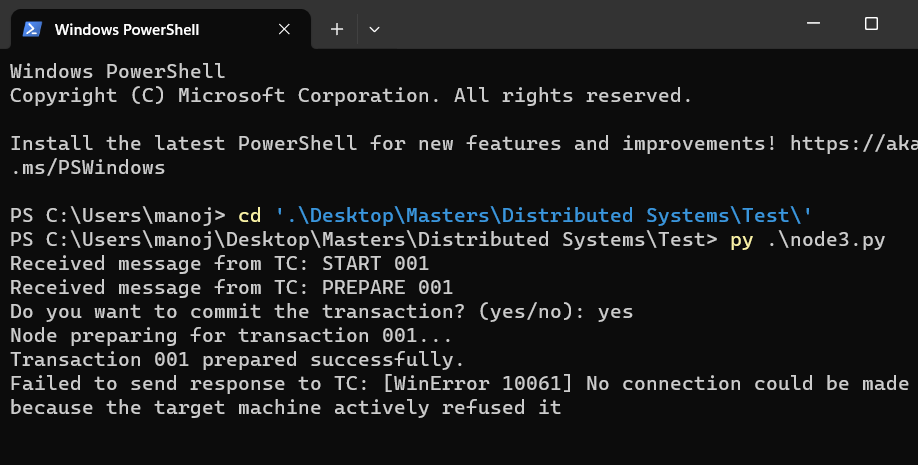
This part of the project demonstrated the system's capability to handle scenarios where the TC fails before sending the 'prepare' message. The implementation ensures that the participant nodes can autonomously decide to abort the transaction in the absence of communication from the TC, maintaining the consistency and integrity of the system.



Transaction coordinator during executing part1



Partcipant node 1, Timed out expecting a prepare message from coordinator. When prepare message came from TC in later point of time, It replies NO to the prepare message irrespective of user choic committing the transaction.



Participant node2 during part1. As one node sent No to the transaction already. TC aborted the transaction. So we see a connection error as the transaction is no longer valid.

**Part 2: Transaction Abortion on 'No' Responses or Response Timeouts**

6.1 Implementation Strategy

**Scenario Description**

This section focuses on scenarios where the Transaction Coordinator (TC) either receives a 'no' response from any participant node or encounters a timeout while awaiting responses during the voting phase of the 2PC protocol.

**Approach**

1. **Voting and Response Collection**: Upon sending the 'prepare' message, the TC awaits 'yes' or 'no' responses from participant nodes within a predefined timeout period.
2. **Handling Negative and Timed-out Responses**:
   * If any participant node sends a 'no' response, the TC immediately aborts the transaction.
   * In case of a timeout (i.e., failure to receive all responses within the specified period), the TC also aborts the transaction.
3. **Internal Abort Decision**: Unlike traditional 2PC implementations, the TC does not send an 'abort' message to the participant nodes. Instead, it internally marks the transaction as aborted and logs this decision on the TC's prompt.
4. **Transaction Logging**: The TC maintains a log of its decisions, including the abort actions taken in response to 'no' responses or timeouts. This log is crucial for analyzing the system's behavior and ensuring transparency in its decision-making process.

6.2 Testing and Results

**Test Setup**

* **Simulating 'No' Responses and Timeouts**: The system was tested under conditions where participant nodes respond with 'no' or do not respond within the timeout period after the 'prepare' message is issued.

**Results**

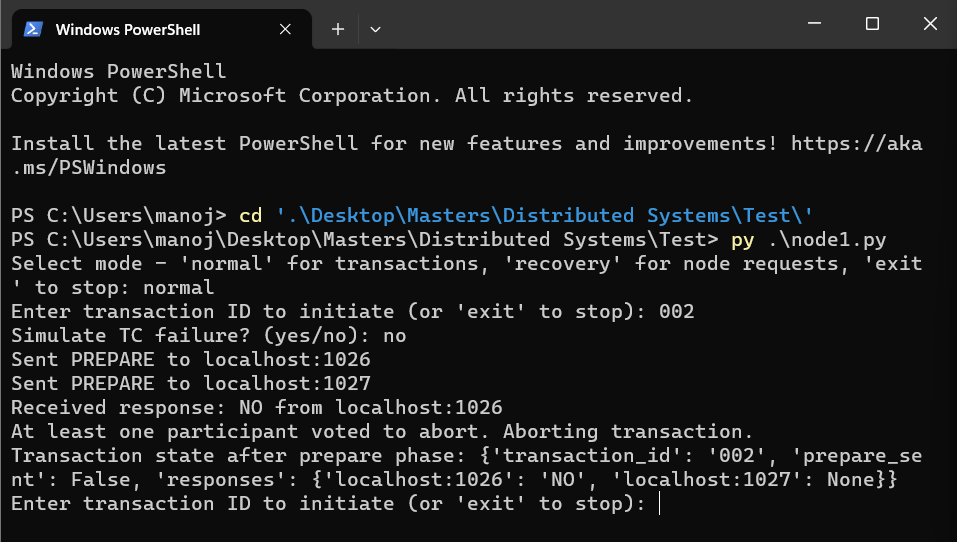
* **Abort on 'No' Response**: The TC promptly aborted transactions whenever it received a 'no' vote from any participant node.
* **Abort on Timeout**: Transactions were also aborted by the TC when not all responses were received within the timeout period.
* **TC Logging**: In both scenarios, the TC correctly logged the abortion of the transaction on its prompt, without sending 'abort' messages to participant nodes.

**Observations**

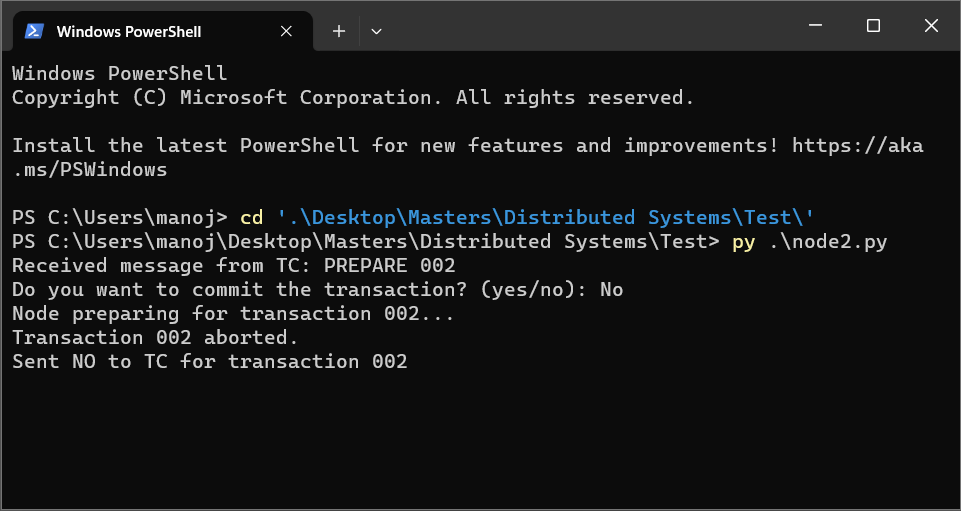
* **Efficiency in Handling Disagreement and Delays**: The TC efficiently handled both explicit refusals and implied refusals (due to timeouts), thus maintaining the integrity of the transaction process.
* **Role of Internal Logging**: The TC's internal logging of abort decisions played a key role in maintaining a record of transaction outcomes, contributing to system transparency and auditability.

**Conclusion**

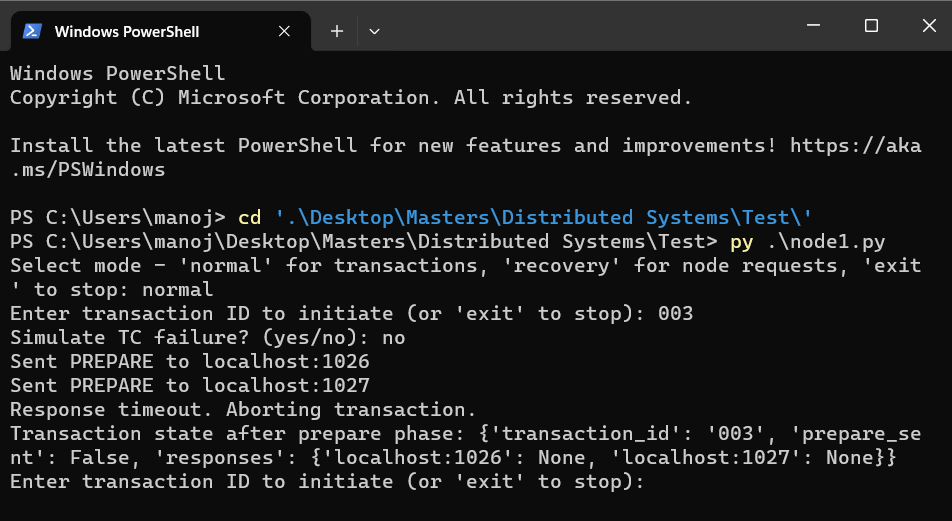
In Part 2, the system demonstrated its capability to effectively abort transactions when unanimous agreement for a commit is not reached, either due to explicit 'no' responses or due to response timeouts. This approach, emphasizing internal decision-making and logging without explicit abort messages to nodes, ensures the maintenance of transaction atomicity and system consistency under varying conditions of participant response.



Transaction coordinator when nodes reply with NO from one of the participant node



Participant node sending NO to prepare message



TC aborting transaction when nodes donot reply for prepare messages

**Part 3: TC Handles Incomplete Commit**

7.1 Implementation Strategy

**Scenario Description**  
This section explores the TC's ability to recover from an incomplete commit scenario, simulated by introducing a sleep timer in the TC code. This method mimics a TC crash during the commit phase, resulting in only partial completion of the commit process.

**Approach**

1. **Simulating Incomplete Commit with Sleep Timer:** After a transaction reaches the commit phase and participant nodes agree to commit, we introduce a sleep timer in the **send\_commit\_messages** function of the TC. This simulates a failure scenario where the TC sends a commit message to one node and then enters a sleep state, representing a crash before it can complete the process with all nodes.
2. **TC Recovery Process:** Upon restart, the TC automatically scans for incomplete transactions. It identifies the 'pending' status in the transaction file, indicating an incomplete commit.
3. **Completion of Partially Committed Transactions:** The TC then resumes the process, sending commit messages to the nodes that are still in the 'pending' state. This step completes the commit process for all participant nodes.
4. **Transaction State Persistence and Logging:** The system maintains a record of the transaction state in the file. The logs from the TC and participant nodes provide transparency and are vital for analyzing the recovery process.

7.2 Testing and Results

**Test Setup**

* **Initiating a Transaction with a Simulated Failure:** A transaction is initiated, and during the commit phase, the TC enters a sleep state (simulated failure) after sending the commit message to one node.
* **Monitoring TC Recovery and Transaction Completion:** After the TC is restarted, its automatic recovery and actions to complete the transaction are monitored.

**Results**

* **Successful Identification of Incomplete Commit:** The TC correctly identified the incomplete commit from the transaction file upon restarting.
* **Effective Completion of the Transaction:** The TC successfully sent commit messages to the 'pending' nodes, ensuring full commitment across all participant nodes.
* **Consistency and Integrity Maintenance:** The final state of the transaction file and node logs confirmed the successful completion of the commit, maintaining the distributed system's integrity.

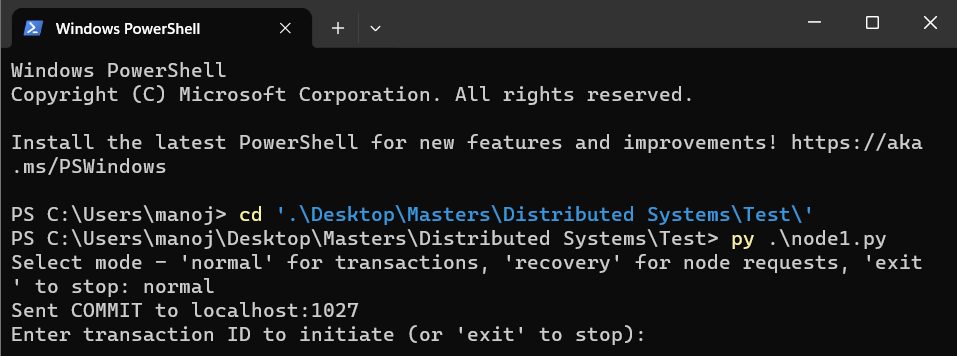
**Observations**

* **Robustness of the Recovery Mechanism:** The TC's ability to detect and complete partially committed transactions demonstrates a robust recovery mechanism in real-world scenarios.
* **Effectiveness of Simulated Failure:** The use of a sleep timer to simulate TC failure provides a more realistic test environment and demonstrates the system's resilience.

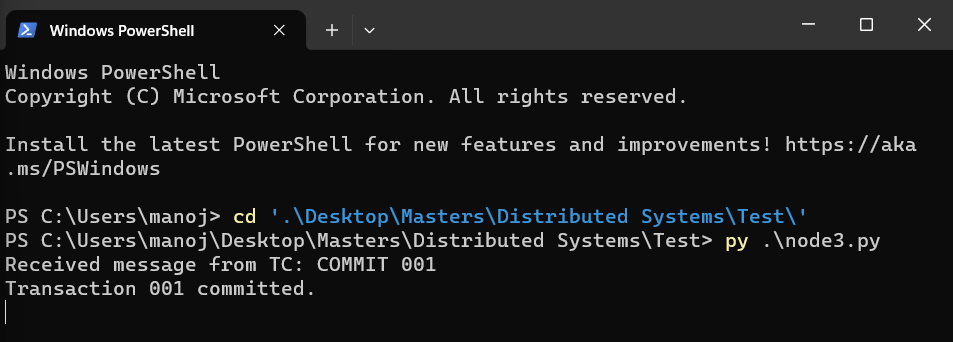
**Conclusion**  
Part 3 demonstrates the TC's capability to recover from a simulated failure during the transaction commit phase. This test underscores the importance of the TC's role in managing incomplete transactions and maintaining the consistency and integrity of distributed systems.

**Example of Transaction File Status After Simulated Failure and Recovery:**

* Post Failure: **001,localhost:1026:done|localhost:1027:pending**
* After TC Restart and Recovery: **001,localhost:1026:done|localhost:1027:done**



After TC restart it immediately identifies 001 commit message is not being sent to node 2 and it sends it to the node having pending status in transaction file.



Participant received it.

**Part 4: Node Recovery After Timeout and 'Yes' Confirmation**

8.1 Implementation Strategy

**Scenario Description**  
This part focuses on a scenario where a participant node sends a 'yes' vote to the Transaction Coordinator (TC), but experiences a failure before receiving the final commit or abort decision. The objective is to test the node's recovery mechanism, ensuring it can successfully inquire and align with the TC's decision upon restart.

**Approach**

1. **Simulated Failure Post-'Yes' Vote:** A participant node sends a 'yes' vote in response to the TC's prepare request. The node then simulates a failure (e.g., crash) using a sleep mechanism and subsequent manual stop.
2. **Node Restart and Recovery Process:** Upon restart, the participant node checks its 'aborted commits' file for any transactions in progress at the time of its failure.
3. **Inquiry to TC:** The node sends an 'INQUIRE' message to the TC for each transaction listed in its 'aborted commits' file.
4. **TC’s Response to Node Inquiries:** The TC, upon receiving an inquiry, checks the transaction's status (committed or aborted) and responds to the node accordingly.

8.2 Testing and Results

**Test Setup**

* **Initiating the Transaction with a Simulated Failure:** A transaction is initiated, and one participant node responds with 'yes'. The node then simulates a failure after this response.
* **Node's Recovery Action:** The node restarts and checks its 'aborted commits' file for transactions needing status clarification.

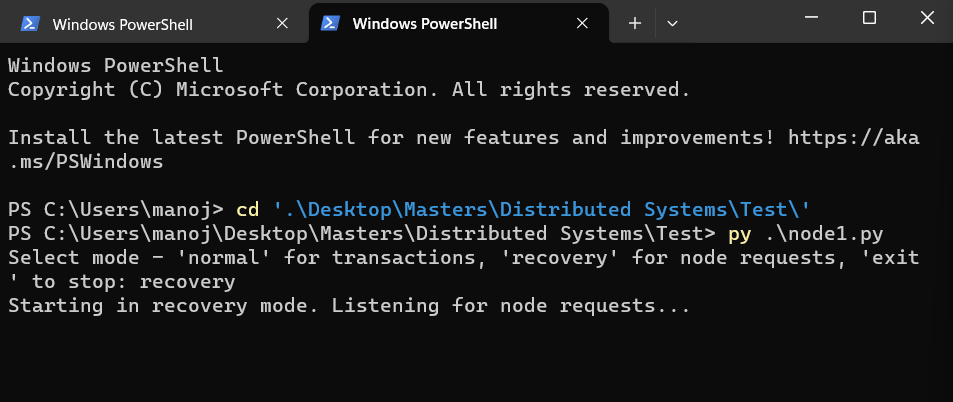
**Results**

* **Node Recovery and Inquiry:** Upon restart, the node inquired about the status of the transaction from the TC.
* **Accurate Response from TC:** The TC responded with the transaction's actual status (commit or abort).
* **Consistency and Integrity Maintenance:** The participant node followed through with the TC's instructions, thereby maintaining the integrity of the transaction process.

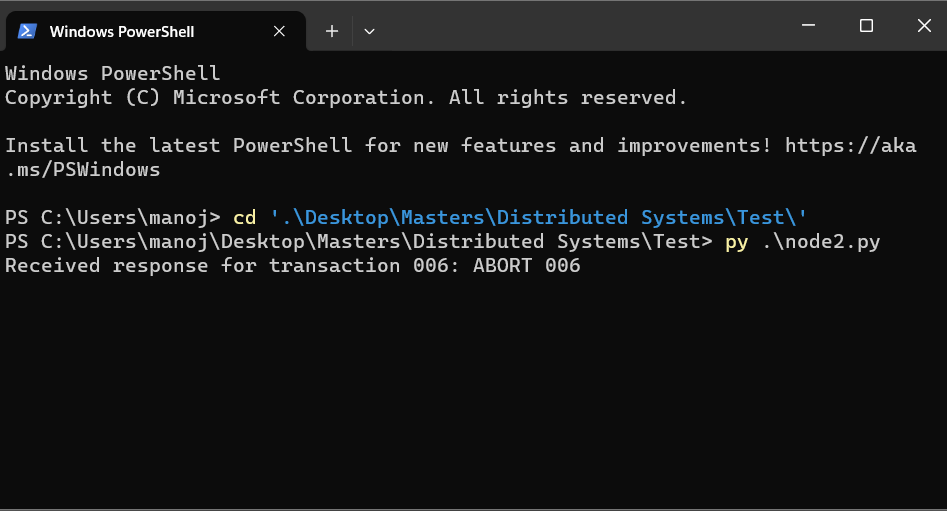
**Observations**

* **Resilience in Handling Failures:** The system effectively handled a failure scenario where a node crashes after agreeing to a transaction.
* **Robust Recovery Mechanism:** The node's ability to inquire about the transaction status upon restart and reconcile its state based on the TC's response demonstrated a robust recovery mechanism.
* **Importance of Persistent State Management:** The use of a persistent 'aborted commits' file was crucial for the node's recovery process.

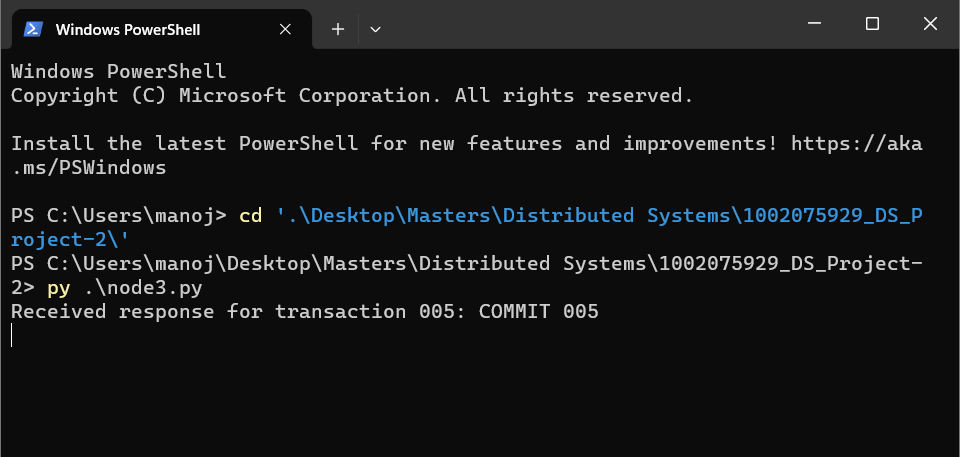
**Conclusion**  
Part 4 illustrates the system's ability to handle scenarios where a participant node fails after agreeing to a transaction but before receiving the final decision. The recovery mechanism, which includes node restarts, inquiries, and accurate responses from the TC, ensures that the system maintains transaction integrity and consistency, even in the face of node failures. This part of the project underscores the resilience and fault-tolerance of the distributed commit process.



TC working on recovery mode for responding to INQUIRE messages for participant node recovery



As soon as node is restarted. It finds 006 transaction in list of incomplete transactions and send INQUIRE message to TC for which TC replies with abort 006 as it is aborted transaction. If the transaction is commited it replies with COMMIT 006



Similary I Stopped node3 with sleep timer when node is back up it sends INQUIRE message for 005 to TC and receives COMMIT message for that respective transaction.

**9. Failure Simulation and Handling Techniques**

Methods Used for Simulating Failures

1. **Simulated TC Failure**: Deliberately preventing the TC from sending the 'prepare' message, thereby simulating a TC failure before the voting phase.
2. **Participant Node Timeout**: Creating a scenario where a participant node does not respond within a specified timeout, triggering an abort decision by the TC.
3. **Manual Transaction File Modification**: Editing the transaction file to simulate an incomplete commit scenario, where the TC has partially committed a transaction.
4. **Simulated Node Crash**: Simulating a crash of a participant node after sending a 'yes' vote but before receiving the commit message.

Handling Techniques

* **Transaction Persistence**: Storing transaction states in files to facilitate recovery and completion of transactions in case of TC failure.
* **Recovery Mode in TC**: Implementing a recovery mode in the TC to handle inquiries from participant nodes about the status of transactions post-recovery.
* **Automated Recovery Process**: Enabling participant nodes to automatically check their 'aborted commits' file upon restart and inquire about the transaction status from the TC.

**10. Results and Analysis**

Test Outcomes

* **TC Failure Before 'Prepare'**: Participant nodes correctly aborted transactions due to the absence of the 'prepare' message, demonstrating effective failure detection.
* **Handling Missing 'Yes' Responses**: The TC successfully aborted transactions when not all participant nodes agreed to commit, ensuring consistency.
* **Incomplete Commit Recovery**: The TC efficiently recovered from simulated incomplete commit scenarios, ensuring the completion of transactions.
* **Node Recovery Post-Crash**: Participant nodes effectively inquired about transaction status post-recovery and aligned their actions with the TC’s response.

System Behavior Analysis

* **Fault Tolerance**: The system exhibited high fault tolerance, effectively managing various failure scenarios and ensuring transaction consistency.
* **Resilience**: The recovery mechanisms for both the TC and participant nodes proved robust, allowing for consistent completion of transactions even in adverse conditions.
* **Consistency Maintenance**: Despite different failure scenarios, the system maintained consistency, adhering to the atomic nature of transactions.

**Final Conclusion and Findings**

The project successfully implemented and tested a fault-tolerant Two-Phase Commit protocol in a distributed system. The system demonstrated a high degree of resilience and consistency, effectively handling various simulated failure scenarios, including TC failures, participant node crashes, and timeouts. The implementation of recovery mechanisms in both the TC and participant nodes played a crucial role in maintaining the integrity and consistency of the transaction process, even in the face of unexpected failures.

**Challenges Faced**

1. **Simulating Realistic Failures**: Creating failure scenarios that realistically mimic network and system failures presented a significant challenge.
2. **Synchronization and Timing**: Ensuring proper synchronization and timing, especially in handling timeouts and coordinating messages between the TC and participant nodes, was complex.
3. **Consistency Across Nodes**: Maintaining transaction consistency across different nodes in various failure scenarios required careful design and testing.
4. **Error Handling and Recovery**: Implementing robust error handling and recovery mechanisms to deal with unpredictable network behavior and system crashes was a critical and challenging aspect of the project.

Top of Form