Distributed Systems

Project 2

I have neither given nor received unauthorized assistance on this work.

Sign: Harsh Shah (1002057387) Date: 11/17/2024

Sign: Vashishth Gajjar (1002160256) Date: 11/17/2024

Under **Professor Mohammed Zaman Noor**

# Overview

This project investigates the deployment of a fault-tolerant two-phase distributed commit protocol, concentrating on managing node failures within a distributed system that includes a single transaction coordinator and multiple participants. We explore a range of failure scenarios, such as crashes of the coordinator and the recovery of participants, to gauge the robustness of the 2PC protocol. Our research provides valuable insights into the management of distributed transactions and the overall stability of such systems, highlighting the importance of fault tolerance in distributed computing environments.

# Part-1 TC Failure Before Sending 'Prepare' Message

In this phase, We developed and analyzed the behavior of a fault-tolerant 2-Phase Commit (2PC) protocol, focusing on a scenario where the Transaction Coordinator (TC) fails before distributing the 'prepare' message. The TC, responsible for initiating the transaction and collecting votes from participants, was designed to handle sudden failures effectively.

* **Simulating TC Failure**: If the TC fails prior to sending out the 'prepare' message, participants will remain in a waiting state until their timeouts are triggered, leading them to abort the transaction. This timeout mechanism was critical to ensuring participants would not stay indefinitely in a pending state, enhancing the system's reliability.
* **Post-Recovery Behavior**: When the TC recovers and attempts to restart the transaction by resending the 'prepare' message, participants that have already timed out will respond with a 'no' vote due to their abort state. This dynamic showcased how the 2PC protocol handles early-stage coordinator failures and prevents indefinite waits.
* **System Resilience**: Implementing and testing this scenario highlighted the importance of a robust timeout mechanism for both the TC and participants to handle unexpected failures gracefully and maintain consistency.

**Functionalities:**

1.**Node Class:**

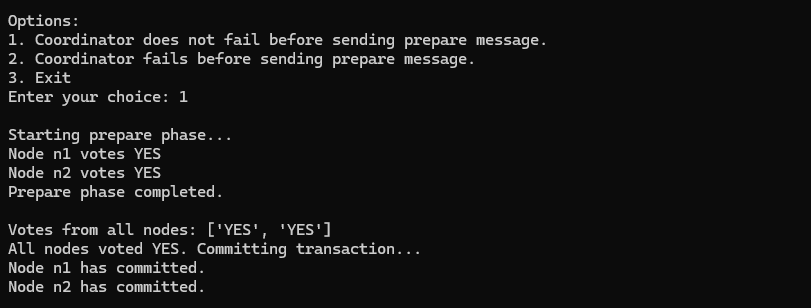
* \_\_init\_\_(self, node\_id): Initializes a node with an ID and a vote state (None initially).
* prepare\_vote(self, coordinator\_failure): Simulates the voting process during the 'prepare' phase. If a coordinator failure is simulated, the node times out and votes "NO". Otherwise, it votes "YES" with a 75% probability or "NO" with a 25% probability.
* commit\_transaction(self): Simulates the commit process and prints a message indicating the node has committed.
* abort\_transaction(self): Simulates the abort process and prints a message indicating the node has aborted.

2. **TransactionCoordinator Class:**

* \_\_init\_\_(self, nodes): Initializes the coordinator with a list of participant nodes.
* simulate\_coordinator\_failure(self): Simulates the failure of the coordinator before sending the 'prepare' message by pausing the process for 5 seconds to indicate downtime, then prints a recovery message.
* initiate\_prepare\_phase(self, coordinator\_failure): Begins the 'prepare' phase by triggering the prepare\_vote method on each node. If a coordinator failure is chosen, it calls simulate\_coordinator\_failure() before proceeding.
* finalize\_commit\_phase(self): Collects votes from all nodes. If all votes are "YES", it commits the transaction; otherwise, it aborts. Nodes commit or abort based on this decision.

3. **run\_simulation() Function**:

* Displays options for the user:
  + Option 1: Run the protocol without coordinator failure.
  + Option 2: Simulate a coordinator failure before sending the 'prepare' message.
  + Option 3: Exit the simulation.
* Based on user input, it creates nodes, instantiates a TransactionCoordinator, and runs the prepare and commit/abort phases.



A screen shot of a computer error

Description automatically generated

# Part-2 TC Aborts Due to Missing 'Yes' Votes

# In the next scenario, We modeled the behavior of the 2PC protocol when the TC does not receive unanimous 'yes' votes from the participants, leading it to abort the transaction.

# **Voting Logic and Decision Making**: The TC starts the transaction by broadcasting the 'prepare' message to all participants. Each participant evaluates its state and replies with either a 'yes' or 'no' vote. If the TC receives even one 'no' vote or fails to receive a response due to a timeout, it makes the decision to abort the transaction.

# **Abort Communication**: The TC then sends 'abort' messages to all participants, who log their state as 'aborted' and output a message indicating they are rolling back. This communication prevents participants from committing to a transaction that is not fully agreed upon, thereby preserving data consistency across the distributed system.

# **System Insight**: This part demonstrated the importance of unanimous agreement in 2PC and emphasized how the protocol prioritizes consistency and reliability by ensuring that a single dissent results in an abort.

**Functionalities:**

1. **Node Class:**

* \_\_init\_\_(self, node\_id): Initializes a Node instance with a unique identifier (node\_id) and sets its initial vote state to None.
* prepare\_vote(self, timeout=5): Simulates the voting process during the 'prepare' phase:
  + Introduces a delay to mimic response time.
  + Randomly simulates non-responses with a probability of 20%.
  + Sets the node's vote to "YES" (75% chance) or "NO" (25% chance) if it responds.
* commit\_transaction(self): Simulates the process of committing a transaction, with a delay for realism.
* abort\_transaction(self): Prints a message indicating the node has aborted the transaction.

2**. TransactionCoordinator Class:**

* \_\_init\_\_(self, nodes): Initializes a TransactionCoordinator instance with a list of participant nodes.
* initiate\_prepare\_phase(self):
  + Starts the 'prepare' phase, creating and running a separate thread for each node to simulate concurrent voting.
  + Waits for all threads to complete before concluding the phase.
* finalize\_commit\_phase(self):
  + Collects and prints the votes from all nodes.
  + Decides whether to commit or abort the transaction:
    - Aborts if any node failed to respond (None in votes) or if any node voted "NO".
    - Commits if all nodes voted "YES".
* \_abort\_all\_nodes(self): Helper method to send an abort command to all nodes.
* \_commit\_all\_nodes(self): Helper method to send a commit command to all nodes.

3**. run\_simulation() Function:**

* Creates a list of participant nodes (Node instances).
* Initializes a TransactionCoordinator with these nodes.
* Runs the 'prepare' and 'commit/abort' phases sequentially to simulate the 2PC protocol.

# 

# Part-3 TC Failure After Sending Some Commit Message

# This scenario explored the behavior of the 2PC protocol when the TC fails after sending a 'commit' message to some participants but before others receive it.

# **Commit Phase Logging**: The TC was programmed to store transaction details persistently before broadcasting the 'commit' message. This ensured that if the TC failed during the commit phase, it could resume and complete the transaction upon recovery.

# **Post-Failure Protocol**: Upon restarting, the TC checked its logs and determined that it needed to resend the 'commit' message to participants that had not acknowledged it. This allowed the TC to ensure that all participants reached a consistent transaction state.

# **Participant Response**: Participants that had not previously received the 'commit' message accepted it upon the TC's recovery and moved to a 'committed' state. This demonstrated how the TC could handle partial commit distribution failures, ensuring data consistency despite disruptions.

**Functionalities:**

**1. Node Class:**

* **\_\_init\_\_(self, node\_id)**: Initializes a Node instance with a unique identifier (node\_id) and sets its initial state to "INIT".
* **commit\_transaction(self)**: Simulates committing a transaction by changing the node's state to "COMMITTED" and printing a confirmation message.

**2. TransactionCoordinator Class:**

* **\_\_init\_\_(self, nodes)**: Initializes the TransactionCoordinator with a list of nodes and sets up the log file (tc\_log.txt) for transaction states.
* **log\_transaction(self, message)**: Logs a transaction state (e.g., a commit message) to the log file, appending a new line with each entry.
* **recover(self)**: Reads the log file and identifies nodes that have already received a commit message before a failure. It updates self.recovered\_commit\_nodes to reflect these nodes and prints the list of recovered nodes.
* **finalize\_commit\_phase(self)**: Finalizes the commit process by sending commit messages to all nodes that did not receive a commit before the failure. Logs each new commit and triggers the commit\_transaction method for those nodes.

**3. run\_simulation() Function:**

* **Creates nodes**: Initializes a set of nodes (Node("n1"), Node("n2")).
* **Starts the commit phase**: The coordinator logs a commit for one node and instructs it to commit.
* **Simulates a failure**: Introduces a pause (time.sleep(5)) to simulate the coordinator's failure after partially committing.
* **Recovers the coordinator**: Triggers the recovery process, reading from the log file to identify already committed nodes.
* **Finalizes the commit**: The coordinator resumes the commit phase, ensuring any remaining nodes complete the commit process.

# 

# Part-4 Participant Recovery After Voting 'Yes'

# In this final scenario, We examined how a participant node responds when it fails after voting 'yes' but before receiving the final commit or abort message from the TC.

# **Pre-Failure State Logging**: Before sending a 'yes' vote, participants recorded their state. This logging was crucial for enabling participants to recover to a 'prepared' state if they failed after voting.

# **Recovery Process**: Upon recovery, the participant read its state and recognized that it had committed to the transaction but had not completed it. The participant then contacted the TC to retrieve the final decision, ensuring it moved to the correct state (either 'commit' or 'abort').

# **Synchronization with TC**: The participant's ability to synchronize with the TC upon recovery showcased the protocol’s ability to maintain data consistency and coordination across the distributed system, even when nodes experienced mid-transaction failures.

**Functionalities:**

**1. Node Class:**

* **\_\_init\_\_(self, node\_id)**: Initializes a Node instance with a unique ID, vote state (None initially), transaction information, and a committed status.
* **prepare\_vote(self, transaction\_id)**: Simulates the voting process for a given transaction. Each node votes "YES" with a 75% probability; otherwise, it votes "NO". The vote is printed, and transaction information is stored.
* **commit\_transaction(self)**: Simulates the commit process by marking the node as committed and printing a commit confirmation.
* **abort\_transaction(self)**: Simulates an abort process by printing an abort message.
* **recover(self, transaction\_id)**: Simulates node recovery after a failure. If the node voted "YES" but hasn't committed, it restores transaction information and prints a recovery message.

**2. TransactionCoordinator Class:**

* **\_\_init\_\_(self)**: Initializes the coordinator with an empty list for threads.
* **initiate\_prepare\_phase(self, transaction\_id, nodes)**: Starts the prepare phase by initiating the voting process for all nodes. Each node's prepare\_vote method runs in its own thread, and the coordinator waits for all nodes to finish voting.
* **finalize\_commit\_phase(self, nodes)**: Determines the final phase by analyzing votes. If all nodes vote "YES", it commits the transaction for all nodes. If any node votes "NO", it aborts the transaction for all nodes.

**3. run\_simulation() Function:**

* **User Interaction**: Provides options to simulate node recovery after a failure or exit the simulation.
* **Transaction Initialization**: Creates nodes and a coordinator and assigns a transaction ID ("T1").
* **Prepare Phase Simulation**: Initiates the prepare phase and waits for all nodes to vote.
* **Node Failure and Recovery**: Simulates a failure, pauses for a few seconds, and then triggers node recovery, where each node checks and restores its transaction state if needed.
* **Commit Phase**: Finalizes the transaction based on node votes—either commits or aborts.

# 

# Learnings

* **Understanding of the Two-Phase Commit Protocol**: Gained a comprehensive understanding of how the 2PC protocol works in distributed systems, emphasizing the importance of achieving consensus for transaction safety.
* **Concurrency in Python**: Learned to implement multi-threaded simulations using Python’s threading module to mimic real-world concurrent operations among distributed nodes.
* **Fault Tolerance Mechanisms**: Developed skills in designing systems with fault tolerance by simulating coordinator and node failures and handling their recovery processes.
* **Persistent Logging for Recovery**: Understood the necessity of persistent logging to maintain system state, enabling reliable recovery of the coordinator after failures.
* **Node Recovery Processes**: Explored strategies for nodes to recover from failures, ensuring they can restore transaction states to prevent inconsistencies.

# Difficulties

# **Thread Synchronization**: Managing thread synchronization was challenging, especially ensuring that all nodes’ voting threads completed before moving to the commit phase.

# **Simulating Realistic Failures**: Creating realistic failure scenarios that accurately mimicked real-world distributed system issues required careful planning and testing.

# **Ensuring Log Consistency**: Implementing and maintaining a consistent log file for coordinator state recovery was complex, particularly in handling edge cases where the log might be partially written before a failure.

# **Node Recovery Logic**: Designing node recovery to accurately restore state without inconsistencies was tricky, especially when simulating partial transactions and ensuring nodes committed properly post-recovery.

# **Randomized Behavior**: Introducing randomized responses (e.g., non-responses or voting “NO”) added variability but required additional logic to handle and ensure correct protocol behavior