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

Sign: Vinod Kumar Puttamadegowda Date: July 2nd, 2024

Name: Vinod Kumar Puttamadegowda Student ID: 1002028556

Project Report: Fault-Tolerant 2-Phase Commit Protocol

Introduction

The objective of this project was to implement a fault-tolerant 2-Phase Commit (2PC) protocol using Python. This protocol is essential in distributed systems to ensure consistency across multiple nodes when a transaction is processed. The 2PC protocol comprises two phases: the prepare phase and the commit/abort phase. The project was divided into four parts, each focusing on different failure scenarios and handling them appropriately.

Implementation

Part 1: Coordinator Failure Before Sending "Prepare"

- **Objective:** Handle the scenario where the coordinator fails before sending the "prepare" message.
- **Approach:**
  - Implemented the coordinator to send a "prepare" message with a unique transaction ID.
  - Simulated a failure by delaying the coordinator's response.
  - Participants were designed to time out and automatically abort if they did not receive a message within a specified period.
  - Participants responded with "no" upon receiving a delayed "prepare" message after recovery.

Part 2: Aborting Transaction Without "Yes" from All Nodes

- **Objective:** Ensure the transaction is aborted if the coordinator does not receive "yes" from all participants.

- **Approach:**
  - The coordinator waits for "yes" responses from all participants.
  - If any participant responds with "no" or a timeout occurs, the coordinator sends an "abort" message to all participants.
  - The participants update their transaction state to "abort" upon receiving this message.

Part 3: Coordinator Failure After Sending "Commit"

- **Objective:** Handle coordinator failure after it sends a "commit" message to some participants.
- **Approach:**
  - The coordinator logs the transaction state to disk before sending "commit" messages.
  - Simulated failure after sending a commit message to one participant.
  - Upon recovery, the coordinator checks the log and sends commit messages to any participants that did not receive it initially.
  - Ensured that participants commit the transaction if they receive the commit message after recovery.

Part 4: Participant Failure After "Yes"

- **Objective:** Manage participant failure after it replies "yes" to the coordinator.
- **Approach:**
  - Participants log the transaction data before replying "yes" to ensure they can recover after a failure.
  - Simulated participant failure after sending a "yes" message.
  - Upon recovery, participants request the transaction status from the coordinator to determine whether to commit or abort.
  - The coordinator responds with the final decision, allowing participants to complete the transaction.

Lessons Learned

1. **Importance of Logging:**
   - Persistent logging is crucial in distributed systems for recovery after failures. Both the coordinator and participants used logs to store transaction states, enabling them to recover correctly.
2. **Timeouts and Retries:**
   - Implementing timeouts and retry mechanisms is vital to ensure system reliability. Timeouts were used to detect failures, and retry logic helped nodes recover and request the current transaction status.
3. **Concurrency Management:**

- Managing multiple processes (coordinator and participants) required careful synchronization, particularly when simulating failures and ensuring nodes communicate effectively after recovery.

4. **Error Handling:**
    - Handling malformed messages and network errors was essential for robustness. Each node was equipped to detect and manage unexpected message formats and disconnections gracefully.

Issues Encountered

1. **Socket Management:**
    - One significant challenge was handling socket disconnections and ensuring that nodes could recover gracefully. Proper error handling and socket timeouts were necessary to address this.
2. **Synchronization Errors:**
    - Simulating failures and recovery in a controlled environment required precise timing. Initially, participants sometimes did not recover correctly due to synchronization issues between the coordinator and participant processes.
3. **Malformed Messages:**
    - Parsing errors and malformed messages caused issues during implementation. These were resolved by implementing stringent message validation and error-checking mechanisms.
4. **Concurrency Bugs:**
    - Multi-processing introduced concurrency bugs, particularly when accessing shared resources like transaction logs. Implementing locks and ensuring atomic operations resolved these issues.

Conclusion

This project provided valuable insights into implementing a fault-tolerant 2PC protocol, highlighting the complexities of distributed systems. The key takeaway is the importance of robust error handling, persistent logging, and clear communication protocols to ensure consistency and reliability. Through this project, the 2PC protocol was successfully implemented to handle various failure scenarios, ensuring transactions were consistently processed across multiple nodes.