The University of Texas at Arlington

Department of Computer Science Engineering

CSE 5306 – 001 – FALL 2024

DISTRIBUTED SYSTEMS

Assignment 2

Team members:

Abhinay Kotla: 1002195827

Hema Sri Puppala: 1002199993

We have not provided or received any unauthorized assistance on this project and will not share the project description and solution online.

Sign:

Hema Sri Puppala. Date: 11/18/24

Abhinay Kotla.

Distributed Systems Project Report

Fault-Tolerant Two-Phase Commit Protocol (2PC)

Introduction

This project demonstrates a fault-tolerant implementation of the Two-Phase Commit Protocol (2PC) in a distributed system using Python. The project is divided into four parts, each progressively introducing additional failure scenarios and recovery mechanisms to evaluate the robustness of the protocol.

Implementation

Part 1: Manager Timeout Before Sending "Prepare"

Objective: Handle a scenario where the manager delays sending the 'prepare' message, causing participants to timeout.
Implementation:
- The manager introduces a delay before sending 'prepare.'
- Participants have a timeout mechanism to abort if 'prepare' is not received in time.
- Despite the abort, participants wait for the final decision (commit or abort) from the manager.

Key Features:
- Participants ignore delayed 'prepare' messages after transitioning to the abort state.

Outputs:

Manager:

```
[Manager] Waiting for participants to connect...
[Manager] Participant 1 connected from ('127.0.0.1', 63698)
[Manager] Participant 2 connected from ('127.0.0.1', 63699)
[Manager] Simulating failure...
[Manager] Sent 'prepare' to Participant 1
[Manager] Sent 'prepare' to Participant 2
[Manager] Received 'no' from Participant 1
[Manager] Received 'no' from Participant 2
[Manager] At least one participant disagreed. Aborting transaction.
[Manager] Sending final decision 'abort' to Participant 1
[Manager] Final decision 'abort' sent to Participant 1
[Manager] Sending final decision 'abort' to Participant 2
[Manager] Final decision 'abort' sent to Participant 2
```

Client1:

```
[Participant 1] Connected to Manager.
[Participant 1] Waiting for 'prepare' message...
[Participant 1] Timeout waiting for 'prepare'. Aborting transaction.
[Participant 1] Ignored unexpected message: prepare
[Participant 1] Final decision received: abort
[Participant 1] Connection closed.
```

Client2:

```
[Participant 2] Connected to Manager.
[Participant 2] Waiting for 'prepare' message...
[Participant 2] Timeout waiting for 'prepare'. Aborting transaction.
[Participant 2] Ignored unexpected message: prepare
[Participant 2] Final decision received: abort
[Participant 2] Connection closed.
```

Part 2: Abort Due to Participant Timeout or Rejection

Objective: Abort the transaction if any participant responds with 'no' or times out during the 'prepare' phase.

Implementation:

- Participants respond with 'yes,' 'no,' or simulate a timeout based on their configuration.
- The manager aborts the transaction upon receiving a 'no' or timing out while waiting for a response.

-Key Features:

- Ensures the atomicity of the transaction by aborting when consensus cannot be reached.

Outputs:

Manager:

```
[Manager] Waiting for participants to connect...
[Manager] Participant 1 connected from ('127.0.0.1', 63702)
[Manager] Participant 2 connected from ('127.0.0.1', 63708)
[Manager] Sent 'prepare' to Participant 1
[Manager] Sent 'prepare' to Participant 2
[Manager] Received 'yes' from Participant 2
[Manager] Timeout waiting for response from Participant 1. Assuming 'no'.
[Manager] At least one participant disagreed or did not respond. Aborting transaction.
[Manager] Sending final decision 'abort' to Participant 1
[Manager] Final decision 'abort' sent to Participant 1
[Manager] Sending final decision 'abort' to Participant 2
[Manager] Final decision 'abort' sent to Participant 2
```

Client1:

```
[Participant 2] Connected to Manager.
[Participant 2] Received 'prepare' message.
[Participant 2] Simulating timeout. No response sent.
[Participant 2] Final decision received: abort
[Participant 2] Connection closed.
```

Client2:

```
[Participant 1] Connected to Manager.
[Participant 1] Received 'prepare' message.
[Participant 1] Sent response: yes
[Participant 1] Final decision received: abort
[Participant 1] Connection closed.
```

Part 3: Manager Crash After Sending First Decision

Objective: Handle a manager crash after sending the decision to one participant but before sending it to others.

Implementation:

- The manager logs client statuses and the final decision persistently in a log file.
- Upon restarting, the manager recovers its state from the log and sends the decision to remaining clients.
- Participants continue waiting for the final decision, reconnecting to the manager if necessary.

Key Features:

- Robust recovery mechanism for the manager using persistent logs.

Outputs:

Manager:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part3>python manager.py
[Manager] Waiting for clients to connect...
[Manager] Client 1 connected from ('127.0.0.1', 63729)
[Manager] Client 2 connected from ('127.0.0.1', 63730)
[Manager] All clients connected.
[Manager] Sent 'commit' to 127.0.0.1:63729
[Manager] Simulating crash after sending decision to one client.

D:\Studystuff\UTA\Distributed Systems\Ass 2\part3>python manager.py
[Manager] Recovering from crash...
[Manager] Waiting for 1 clients to reconnect...
[Manager] Reconnected with Client at ('127.0.0.1', 63742)
[Manager] Sent 'commit' to 127.0.0.1:63730
[Manager] Transaction recovery complete.
```

Client1:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part3>python client.py 1
[Participant 1] Connected to Manager.
[Participant 1] Connection closed.
[Participant 1] Connected to Manager.
[Participant 1] Final decision received: commit
[Participant 1] Connection closed.
[Participant 1] Transaction complete. Final decision: commit
```

Client2:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part3>python client.py 2
[Participant 2] Connected to Manager.
[Participant 2] Connection closed.
[Participant 2] Connected to Manager.
[Participant 2] Error: [WinError 10054] An existing connection was forcibly closed by the remote host
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Manager not available. Retrying in 5 seconds...
[Participant 2] Connection closed.
[Participant 2] Connected to Manager.
[Participant 2] Final decision received: commit
[Participant 2] Connection closed.
[Participant 2] Transaction complete. Final decision: commit
```

Part 4: Participant Crash After Responding "Yes"

Objective: Handle a participant crash after responding 'yes' to the 'prepare' message but before receiving the final decision.

Implementation:

- Participants log their state persistently before responding 'yes.'
- If a participant crashes, it recovers from its log and reconnects to the manager to fetch the final decision.
- The manager handles reconnecting participants gracefully, ensuring they receive the correct decision.

Key Features:

- Participant recovery ensures no decision inconsistency due to participant crashes.

Outputs:

Manager:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part4>python manager.py
[Manager] Waiting for clients to connect...
[Manager] Client 1 connected from ('127.0.0.1', 63749)
[Manager] Sent 'prepare' to Client 127.0.0.1:63749
[Manager] Received 'yes' from Client 127.0.0.1:63749
[Manager] Client 2 connected from ('127.0.0.1', 63750)
[Manager] Sent 'prepare' to Client 127.0.0.1:63750
[Manager] Received 'yes' from Client 127.0.0.1:63750
[Manager] All clients agreed. Committing transaction.
[Manager] Sending final decision to clients...
[Manager] Reconnected with Client ('127.0.0.1', 63751)
[Manager] Sent 'commit' to Client 127.0.0.1:63749
[Manager] Reconnected with Client ('127.0.0.1', 63752)
[Manager] Sent 'commit' to Client 127.0.0.1:63750
[Manager] Transaction completed.
```

Client1:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part4>python client.py 1
[Participant 1] Connected to Manager.
[Participant 1] Received 'prepare' message.
[Participant 1] Sent response: yes
[Participant 1] Simulating failure...
[Participant 1] Exiting due to simulated crash.
[Participant 1] Connection closed.

D:\Studystuff\UTA\Distributed Systems\Ass 2\part4>   python client.py 1
[Participant 1] Connected to Manager.
[Participant 1] Fetching final decision after recovery.
[Participant 1] Final decision received: commit
[Participant 1] Connection closed.
[Participant 1] Transaction complete. Final state: commit
```

Client2:

```
D:\Studystuff\UTA\Distributed Systems\Ass 2\part4>python client.py 2
[Participant 2] Connected to Manager.
[Participant 2] Received 'prepare' message.
[Participant 2] Sent response: yes
[Participant 2] Connection closed.
[Participant 2] Connected to Manager.
[Participant 2] Fetching final decision after recovery.
[Participant 2] Final decision received: commit
[Participant 2] Connection closed.
[Participant 2] Transaction complete. Final state: commit
```

Learnings

1. Resilience in Distributed Systems:
   - Implementing the 2PC protocol highlighted the importance of handling failures gracefully to ensure consistency and fault tolerance.
   - Persistent logs played a critical role in recovering state after failures.
2. Timeouts and Retries:
   - Timeouts are essential for detecting failures, and retry mechanisms are crucial for ensuring eventual consistency in distributed systems.
3. Synchronization Challenges:
   - Managing concurrent participant and manager processes required careful synchronization, particularly during recovery.
4. Testing Failure Scenarios:
   - Simulating crashes and network delays provided insight into real-world challenges in distributed systems.

Challenges and Issues Encountered

1. Handling Unexpected Messages:
   - Participants occasionally received unexpected messages during recovery due to retries and reconnections. This required robust message validation.
2. Socket Management:
   - Closing and reopening sockets for reconnecting clients introduced complexities, especially when clients crashed midway.
3. Log Consistency:
   - Ensuring the manager and participants maintained consistent logs, even during crashes, required careful design.
4. Simulating Failures:
   - Introducing realistic delays and crashes while preserving expected behavior posed challenges, particularly in Parts 3 and 4.

Conclusion

This project successfully demonstrates a fault-tolerant implementation of the 2PC protocol, capable of handling various failure scenarios. The use of persistent logs, timeouts, and retry mechanisms ensures consistency and resilience in the face of failures. The challenges encountered and resolved during this project provide valuable insights into building robust distributed systems.

Future Work

- Extend the protocol to support more complex scenarios, such as multiple transactions or cascading failures.
- Optimize recovery mechanisms to handle a larger number of participants efficiently.