Lab: Two-Phase-Commit

Time spent: 40 hours (most of it was in figuring out how to write code in RUST)

1. High-Level Approach and Description of Code Implementation

- I started the assignment with the skeleton code provided and spent around 5-6 hours familiarizing myself with Rust and its programming model.
- I took a few more hours to understand the Two-Phase Commit (2PC) protocol, sketching out the solution by re-watching lectures and exploring external videos for better clarity.
- Next, I focused on understanding how IPC (Inter-Process Communication) works. I experimented with basic IPC channels in the run function of the main class, passing and receiving simple "hello_world" messages to test communication.
- Once I understood IPC, I began by writing the coordinator code, followed by the client, participant, and finally the main class to complete the implementation.

Design Considerations:

- In the initial design, the coordinator used a shared sender and receiver for all clients and participants. Later, I restructured the coordinator to use individual channels for each client and participant.
- The redesign aimed to eliminate potential confusion caused by message ordering on a single channel, which could lead to unnecessary aborts. Additionally, it facilitates parallel processing of messages by the coordinator if needed.
- Exit Conditions:
  1. Exit conditions are very important for different components involved in 2PC for graceful termination.
  2. Coordinator: The coordinator is designed to stop accepting any new requests on `CTRL+C` and sends a termination signal to all connected clients and participants.
  3. Client: Client stops the protocol once it receives a termination signal from coordinator.
  4. Participant: Participant stops the protocol once it receives the termination signal from coordinator.
  5. The above exit conditions, address for completion of any inflight requests once `CTRL+C` is sent from the terminal.
- Timeouts:
  1. I introduced timeouts in the coordinator class during Phase 1 of the 2PC protocol, where the coordinator collects votes from participants. This part of the coordinator logic took more time and effort compared to the rest of the class.
  2. Initially, I implemented a custom timeout mechanism because try_recv_timeout was not available in version 0.14.x of the IPC crate.
  3. Later, I upgraded the IPC crate to the latest version and replaced the custom timeout logic with try_recv_timeout, expecting a slight performance improvement. However, the change did not result in any significant performance gains.

    4. The timeout is set to 25ms, ensuring the coordinator doesn’t wait indefinitely for participant responses and can move on to the next request efficiently.
- -S and -s arguments:
    1. The code simulates unreliable communication between the coordinator and participants using a probabilistic send function. A random value is generated to mimic message loss, and if this value exceeds a predefined probability threshold, the system logs the failure, simulating a dropped message.
    2. Additionally, the code simulates operation failures in a similar way, using probabilistic calculations to introduce the chance of failure during operations.

Utility of 2PC Implementation:

- The implemented 2PC protocol ensures atomicity — a key feature in distributed transactions — by making sure that either all participants commit or none do.
- In practice, this is critical for maintaining consistency in scenarios where multiple systems or databases are involved in a distributed operation.
- SCENARIOS:
    - Scenario 1: All Participants Vote to Commit: If all participants vote to commit, the transaction proceeds successfully, and the coordinator commits. This ensures atomicity by guaranteeing that the transaction will be visible to all systems involved or none at all.
    - Scenario 2: One or More Participants Vote to Abort: If even a single participant votes to abort, the coordinator will abort the transaction. This prevents partial commits where some systems would reflect changes while others do not, ensuring consistency across the distributed systems.
    - Scenario 3: Timeout or Message Loss: If a participant fails to respond within the timeout (or the message is lost due to probabilistic failure), the transaction will not commit. Instead, it will either be aborted or marked as uncertain. This handles cases where the system could not receive all votes due to network issues or participant failures, preventing inconsistent transaction states.
- The robustness of this implementation helps maintain data integrity and consistency across distributed systems, which is critical in use cases like distributed databases, financial transactions, or microservices.

2. Performance gain of code implementation

Efficient Timeout and Error Handling:

- The use of timeouts during vote collection ensures that the coordinator does not wait indefinitely for a participant's vote, improving system responsiveness and failure detection.
- Timeouts allow the system to quickly abort or declare a transaction unknown if a participant fails to respond, rather than stalling indefinitely.

Probabilistic Sending:

- The probabilistic send mechanism introduced a layer of realism by simulating message loss, which added robustness testing for the 2PC protocol.

- The system handles message loss gracefully and aborts or retries transactions as needed without impacting overall throughput significantly.

Scalability and Parallelization:

- The implementation shows noticeable differences in performance as the number of participants increases. For example:
    1. With 1 client, 2 participants and 10k requests, the time taken is 45 seconds.
    2. With 1 client, 20 participants and 10k requests, the time taken increases to 90 seconds.
- In my implementation, each participant communicates with the coordinator through its own dedicated channel. The same is true for client-coordinator communication as well, allowing both types of interactions to be easily parallelized.

3. Technical Difficulty Faced and Insight Gained

Ensuring Correctness with Partial Failures:

- The biggest challenge was ensuring that the coordinator made the correct decision even when some participants failed to send their votes.
- For example, if one participant voted to commit and another timed out, the coordinator needed to handle this gracefully and either abort or mark the transaction as uncertain.
- Ensuring that the coordinator does not incorrectly commit in these scenarios required careful status management for votes.

Timeout Handling in Rust:

- Implementing timeouts to ensure correctness was also challenging due to missing corner cases. After a couple of trials, I was able to get it right without excessive aborts.
- The importance of timeouts in distributed systems cannot be overstated. Timeouts prevent indefinite blocking and allow the system to make progress even when some components are unresponsive.
- Figuring out the right value of timeout is equally crucial as well cause - if it is too less, we see huge number of failures, if it is too high, the system performance degrades greatly in the event of a failure scenario.

Atomicity with Fault Tolerance:

- The 2PC protocol's core strength lies in its ability to ensure atomicity even in the presence of failures.
- This project demonstrated the value of 2PC in ensuring that distributed transactions are either fully committed or fully aborted, even when participants fail or messages are lost.

RUST Programming itself:

- At first, it was quite confusing and difficult, writing a single function took a lot of time. Some error message were misleading as well.
- But as I got more hands-on, I stopped making most obvious errors in the code.

Conclusion

- No system is entirely free of failures, so it's crucial to design systems with resilience in mind.
- This means minimizing points of failure, enabling automatic recovery, and ensuring that the system can continue operating with minimal human intervention when issues do arise.
- In my professional experience with Cockroach DB - a distributed relational database, these principles are particularly important because distributed systems are inherently complex and more prone to various types of failures.
- While I had previously read about the Two-Phase Commit (2PC) protocol, actually implementing it was an exciting and enlightening experience.
- It gave me a much deeper understanding of how distributed systems can ensure consistency, atomicity and handle failures during coordinated transactions across multiple nodes.
- Seeing these concepts in action has been immensely valuable in improving my understanding of a 2PC.