Lab 4 CS 380-2PC with Rust

Abraham Zamora

November 20, 2024

Introduction

This lab involved the implementation of the Two-Phase Commit (2PC) protocol ensures atomic transaction processing across distributed participants while handling potential failure scenarios. This system is designed to facilitate robust communication and coordination between clients, participants, and the coordinator, maintaining transactional integrity. The primary focus is on reliability, state management, and efficient error handling, achieved through modular implementation and detailed logging.

My first challenge was to understand how the Rust programming language works. Reviewing the suggested sources (The Rust Programming Language and Rust by Example) and the documentation for the IpcOneShotServer helped. Understanding ownership and borrowing proved to be challenging and this lab presented an opportunity. In addition, I also watched some additional videos regrading 2PC protocol in order to make sure I full understand the concept(`https://www.youtube.com/watch?v=-_rdWB9hN1c&t=733s`). Once I felt I had a good grasp, I proceeded to work on the lab in full. This part of the assignment took me about 10 hours total. I'm going to address how I approached completing the TODO sections for each file, in no particular order.

Addressing the main.rs file

In the main.rs file, the spawn child and connect function establishes interprocess communication (IPC) between the coordinator and child processes. I first created an IpcOneShotServer for the coordinator to receive messages from the child, updates the child's options with the IPC path, and spawns the child process using the current executable. Communication channels are then set up, and the coordinator accepts the connection from the child, establishing bidirectional communication. This design ensures that each child process has a dedicated channel to the coordinator, facilitating efficient message exchange.

The connect to coordinator function enables child processes to establish communication with the coordinator. It creates a new IpcOneShotServer for the child to receive messages, connects to the coordinator using the provided IPC path, and sends the child's transmitter and server name to the coordinator.

The child then accepts the connection from the coordinator, completing the setup of communication channels. This approach allows the child to initiate communication, ensuring that both coordinator and child are synchronized.

In the run function, the coordinator initializes its log path and creates a new Coordinator instance. It then spawns client and participant processes by cloning the coordinator's options, setting the appropriate mode and number, and invoking spawn child and connect. Each child process is registered with the coordinator using client join or participant join. After spawning all processes, the coordinator starts the protocol and, upon completion, terminates all child processes and waits for them to exit. This structured approach ensures that all components are properly initialized, coordinated, and terminated, maintaining the integrity of the distributed transaction system.

The run client function initiates the client process by connecting to the coordinator to obtain communication channels. It constructs a new Client instance with the provided options and starts the client protocol, processing the specified number of requests. This function ensures that the client is properly connected and actively participates in the 2PC protocol.

Similarly, the run participant function starts the participant process by establishing communication with the coordinator. It creates a new Participant instance, providing parameters such as log path, success probabilities, and the number of requests. The participant then engages in the protocol, contributing to the distributed transaction process. This function ensures that participants are correctly initialized and integrated into the 2PC workflow.

Overall, the implementation demonstrates a structured approach to implementing a two-phase commit protocol in Rust, leveraging IPC channels for communication between processes and ensuring proper initialization, coordination, and termination of all components involved in the distributed transaction system.

Addressing coordinator.rs file

The coordinator.rs implementation of the Two-Phase Commit (2PC) protocol ensures atomicity and consistency across distributed participants and clients. The participant join and client join functions are essential for setting up communication channels. These functions register participants and clients by associating their identifiers with communication channels (senders and receivers) stored in hash maps. These connections allow the coordinator to relay messages during the protocol's execution effectively.

The protocol function is the centerpiece of the coordinator's role. It begins by iterating through client requests, processing them until the maximum number of allowed requests is reached. For each request received from a client, the coordinator transitions to the ReceivedRequest state and logs the request. It then enters the prepare phase, where it changes to the ProposalSent state and logs the intent to propose the transaction. Subsequently, the coordinator broadcasts the proposal to all participants.

In the voting phase, the coordinator collects responses from participants, tallying commit and abort votes. It employs a timeout mechanism to account for participant failures or delays. If all participants vote to commit, the coordinator transitions to the CoordinatorCommit state; otherwise, it moves to CoordinatorAbort. Both decisions are logged, and the decision is broadcast to all participants. The coordinator also sends the final outcome (commit or abort) to the client that initiated the request.

The protocol function's implementation highlights careful management of communication, state transitions, and failure handling. After processing all requests, the coordinator signals an exit to all participants and clients by sending a CoordinatorExit message and logs this event. The function concludes by invoking report status, which aggregates and displays the number of committed, aborted, and unknown transactions, providing an overall summary of the protocol's execution.

This design emphasizes fault tolerance and adherence to 2PC's atomicity guarantees, ensuring all clients and participants reach a consistent state, even in the presence of communication delays or failures. The detailed logging throughout the process supports transparency and debugging during the protocol's operation.

Addressing client.rs file

The client implementation in client.rs handles the client-side responsibilities of the Two-Phase Commit (2PC) protocol. This includes sending transaction requests to the coordinator, receiving results, and maintaining statistics on transaction outcomes. The protocol function orchestrates the main client behavior by looping through a specified number of requests, sending operations to the coordinator using send next operation, and waiting for the coordinator's response using recv result. Each iteration corresponds to a single transaction lifecycle, ensuring that requests are issued only while the client is in a running state.

The send next operation function constructs a unique transaction identifier (txid) for each request, encapsulating it in a ProtocolMessage with the type ClientRequest. This message is sent to the coordinator via an IPC channel. If the transmission fails, the function logs an error and terminates the client process to maintain protocol consistency. Successful transmissions are logged for debugging and verification purposes.

The recv result function handles the receipt of results from the coordinator, which include ClientResultCommit or ClientResultAbort messages. Based on the message type, the function updates counters for committed, aborted, and unknown transactions. Any unexpected message types are logged as warnings, and failures to receive results result in an error being logged and the client terminating.

To ensure graceful termination, the wait for exit signal function listens for an exit signal from the coordinator, checking the CoordinatorExit message type. While waiting, it logs any communication issues and handles disconnections or

IPC errors appropriately. Once the exit signal is received, the function stops the client process cleanly.

Finally, the report status function aggregates and prints the outcomes of all transactions, providing a summary of the client's performance. This includes counts of committed, aborted, and unknown transactions, aiding in post-execution analysis and debugging. Combined, these functions implement the client-side responsibilities of 2PC, ensuring consistent communication with the coordinator and robust handling of transaction outcomes.

Addressing participants.rs file

The participant.rs file implements the participant-side logic for the Two-Phase Commit (2PC) protocol. Each participant interacts with the coordinator to process transaction proposals, vote on their commit or abort, and act on the coordinator's global decisions. The main functionality is distributed across several methods, ensuring clean communication and proper logging of transaction outcomes.

The send method handles transmitting protocol messages from the participant to the coordinator. It incorporates a probabilistic failure mechanism based on a send success prob value. When a message fails to send due to this probability, it logs the failure and moves on. On successful sends, the method ensures the coordinator receives the intended message. This feature allows testing of scenarios where communication might be unreliable, simulating real-world conditions in distributed systems.

The perform operation method simulates executing the operation specified in the transaction. The success of the operation is determined probabilistically by operation success prob. If the operation succeeds, the method logs a ParticipantVoteCommit message, indicating the participant's willingness to commit. Conversely, if the operation fails, it logs a ParticipantVoteAbort message, signaling that the participant has voted to abort the transaction. The method returns a boolean result reflecting the operation's success or failure, feeding into the overall protocol logic.

The report status method aggregates and reports the participant's statistics, such as the number of transactions successfully committed, aborted, or left in an unknown state. This output is crucial for verifying the correctness and behavior of the 2PC protocol during testing and debugging. By printing these statistics in a consistent format, the method facilitates automated evaluation and analysis of the participant's performance.

The wait for exit signal method waits for a CoordinatorExit message from the coordinator to signal the end of the protocol. While waiting, it retries message receipt and logs any errors encountered. Upon receiving the exit signal, the method cleans up and exits gracefully. This ensures that the participant terminates its process only when explicitly instructed, maintaining protocol consistency and avoiding premature termination.

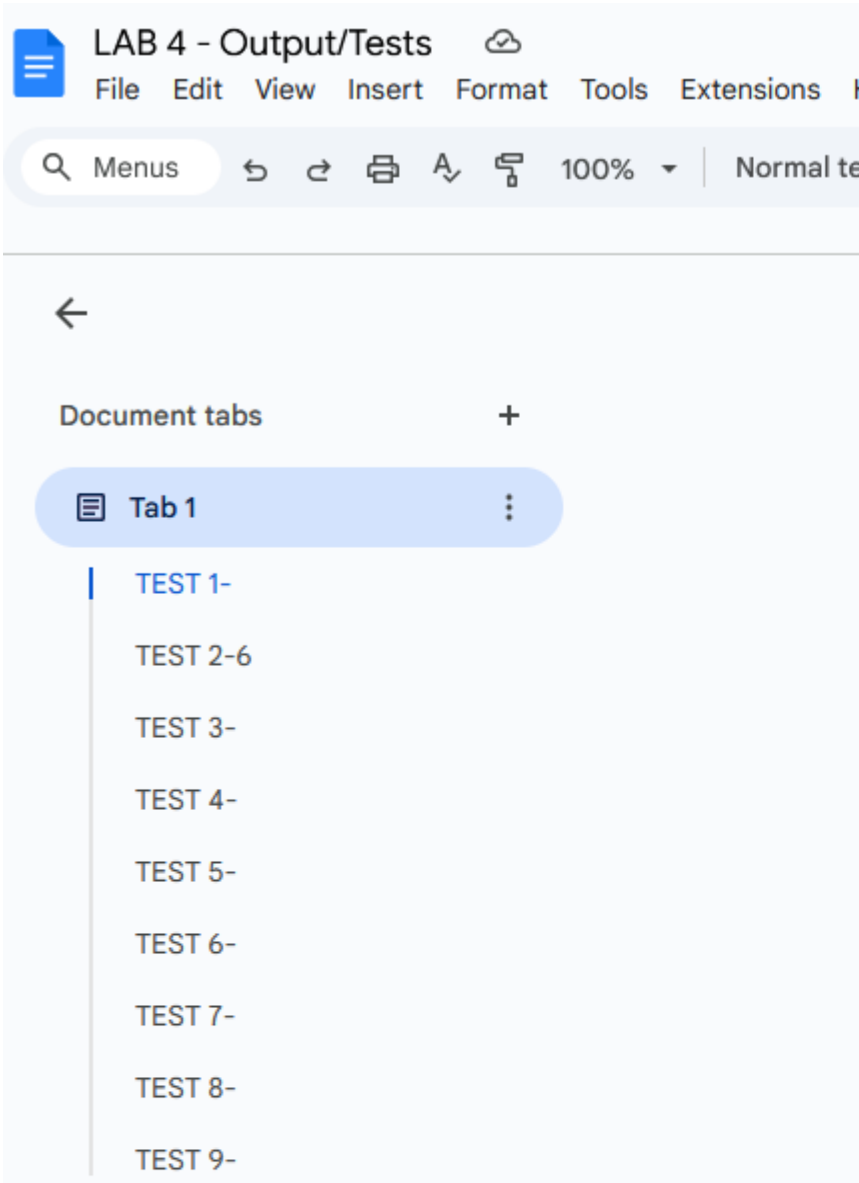
The protocol method drives the participant's main behavior. It continuously

listens for messages from the coordinator while the participant is running. When a CoordinatorPropose message is received, the participant transitions to the ReceivedP1 state, performs the associated operation, and votes by sending either a ParticipantVoteCommit or ParticipantVoteAbort message to the coordinator. If the coordinator subsequently sends a CoordinatorCommit or CoordinatorAbort message, the participant acts accordingly by logging the outcome and updating its state and statistics. The method also gracefully handles CoordinatorExit messages, signaling the end of the protocol. Any unexpected message types are ignored, and the participant retries on message receipt failures to maintain robustness.

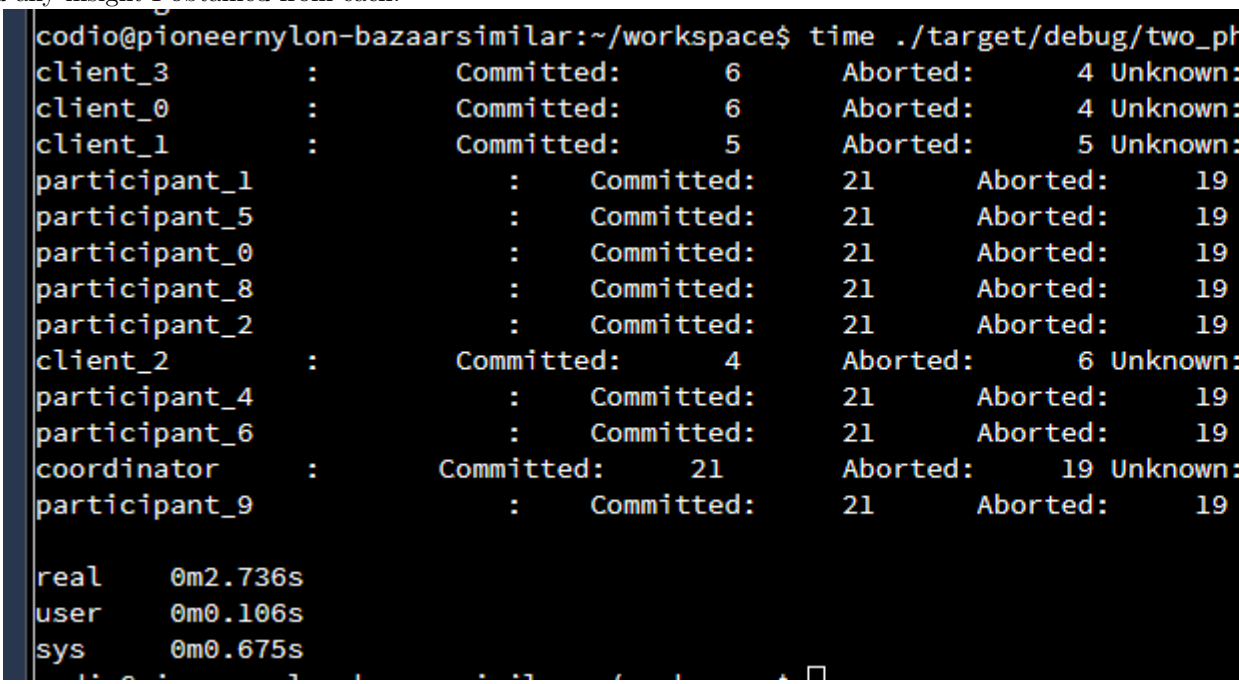
The participant implementation balances simplicity and functionality, using probabilistic failure mechanisms to simulate real-world unreliability in both communication and operations. Logging and state management ensure that each transaction is appropriately tracked and that the protocol can be validated during and after execution. The participant's methods work together to provide a clean and testable implementation of the 2PC protocol.

Testing and observations

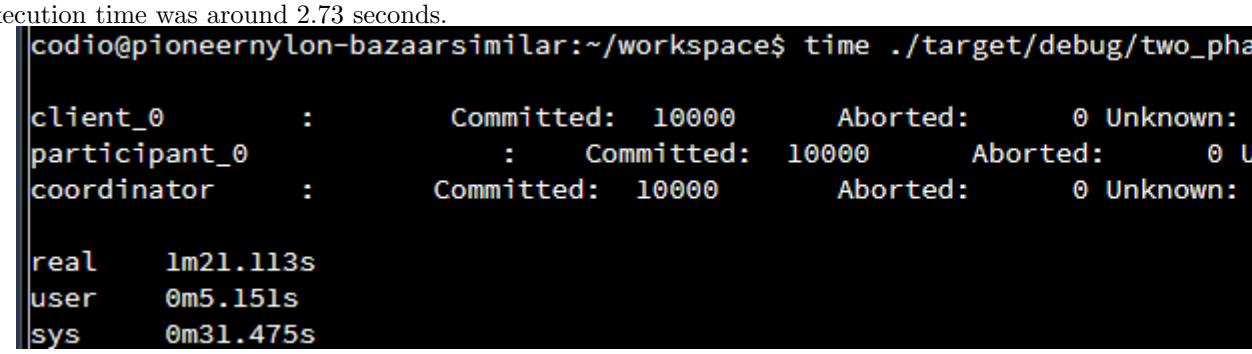
In order to test, I considered a document shared with all of our classmates. The link is included here. (`https://docs.google.com/document/d/16lW9KsKrj4UYOp6zlqn6PTZiM4NY_-lqpCES0v9xYLk/edit?tab=t.0#heading=h.si8e12wbdz7` This page (image included just in case) only has the results of tests that fellow students ran and what


LAB 4 - Output/Tests
File Edit View Insert Format Tools Extensions
Menus
100%
Normal te
Document tabs
Tab 1
TEST 1-
TEST 2-6
TEST 3-
TEST 4-
TEST 5-
TEST 6-
TEST 7-
TEST 8-
TEST 9-

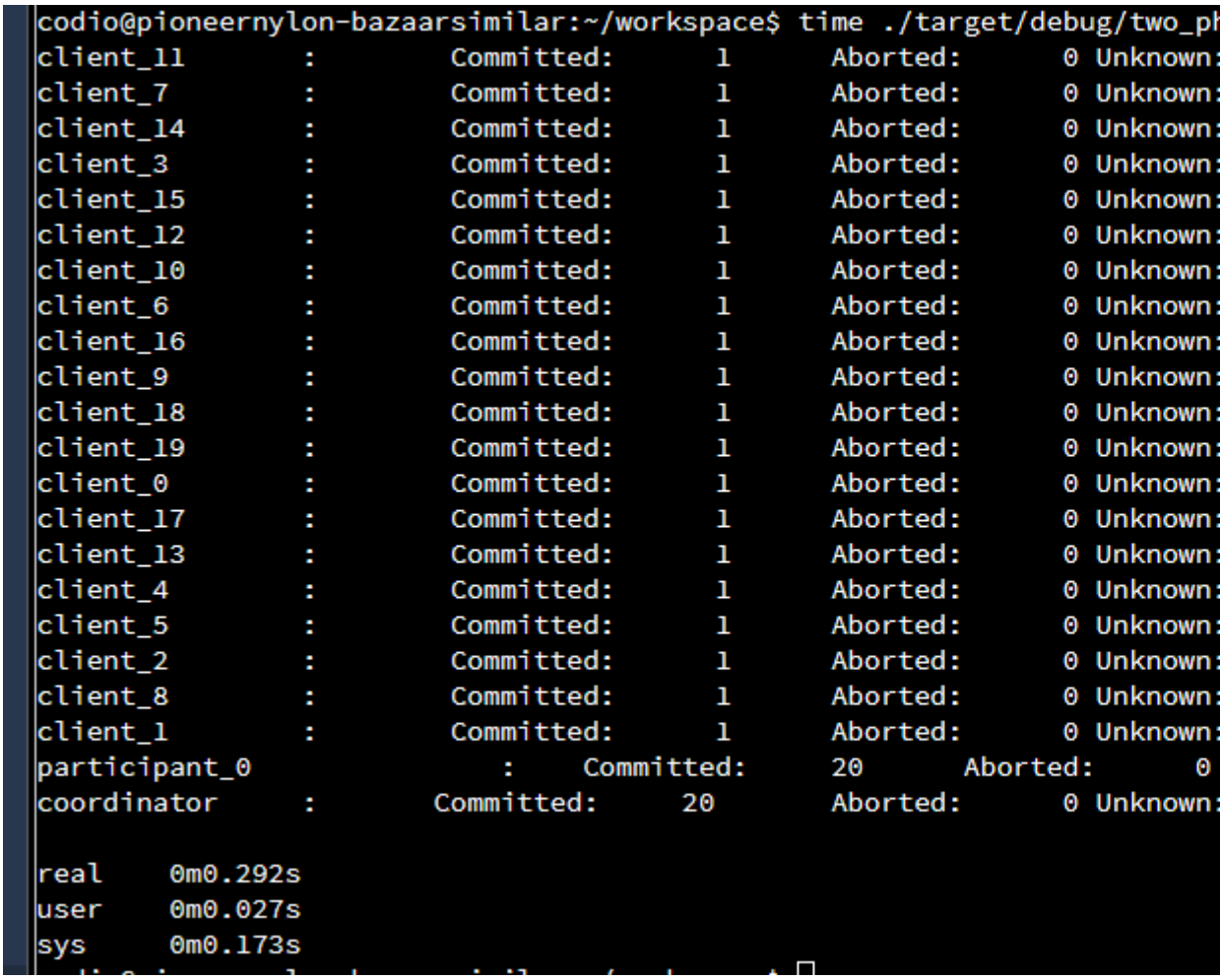
I am including the results I received from various tests and discuss the results and any insight I obtained from each.



The first test I ran called for 4 clients, 10 participants, and 10 requests with s being at .95. Each client committed between 4 to 6 transactions. Participants committed 21 and aborted 19. I re-ran this test a few times and always had between 21 and 30 commits with anywhere between 19 and 10 failures. The execution time was around 2.73 seconds.



The second test was with 1 client, 1 participant, and 10000 requests with both s and S set to 1.0. All 10,000 transactions were committed. This execution time took a little over a minute given the large number of requests. The 1 client and 1 participant here affect the time given that each has to go through 10,000 transactions at once.



Test 3 ran with 20 clients, 1 participant, 1 request with S and s at 1.0. Once again, the success rate was perfect. It executed in around .29 seconds. The simplicity of the setup committing only one transactions results in exceptionally fast execution.

```
codio@pioneernylon-bazaarsimilar:~/workspace$ time ./target/debug/two_phas

client_4        :       Committed:    500      Aborted:      0 Unknown:
client_2        :       Committed:    500      Aborted:      0 Unknown:
client_12       :       Committed:    500      Aborted:      0 Unknown:
client_14       :       Committed:    500      Aborted:      0 Unknown:
client_1        :       Committed:    500      Aborted:      0 Unknown:
client_0        :       Committed:    500      Aborted:      0 Unknown:
client_7        :       Committed:    500      Aborted:      0 Unknown:
client_15       :       Committed:    500      Aborted:      0 Unknown:
client_3        :       Committed:    500      Aborted:      0 Unknown:
client_13       :       Committed:    500      Aborted:      0 Unknown:
client_5        :       Committed:    500      Aborted:      0 Unknown:
client_16       :       Committed:    500      Aborted:      0 Unknown:
client_8        :       Committed:    500      Aborted:      0 Unknown:
client_9        :       Committed:    500      Aborted:      0 Unknown:
client_11       :       Committed:    500      Aborted:      0 Unknown:
client_10       :       Committed:    500      Aborted:      0 Unknown:
client_6        :       Committed:    500      Aborted:      0 Unknown:
client_18       :       Committed:    500      Aborted:      0 Unknown:
client_19       :       Committed:    500      Aborted:      0 Unknown:
client_17       :       Committed:    500      Aborted:      0 Unknown:
participant_0             :    Committed:  10000      Aborted:      0 Un
coordinator     :       Committed:  10000      Aborted:      0 Unknown:

real    1m11.319s
user    0m3.636s
sys     0m25.619s
```

Test 4 ran with 20 clients, 1 participant, 500 requests, and S being set at .95. In this case, the send failed due to the probability check. However, it kept attempting to process. In the end, each client committed around 450 each, aborting around 50 each. The participant committed 9032, aborting 968, and the coordinator registering 497 of the 968 as unknown. The large number of requests over 20 clients ran in a little over a minute.

In general, I noticed that the number of participants and clients impacts the runtime significantly. A single participant can lead to a bottleneck for higher number of transactions. I also messed around in the code a bit in places where the threads sleep for a little bit. Originally, I had them sleep for as long at 500 milliseconds each time. This led to extreme delays. I lowered those sleeps throughout the code to numbers less than 100 milliseconds (and sometimes just a few milliseconds), and that improved the execution time dramatically. Waiting did not really provide any benefits to the number of committed transactions, as

the number really did not change. So waiting so long had no effect in the end.

Conclusion

This lab was a good introduction not just to the 2PC implementation, but also to Rust. The Rust language did provide some initial challenges as I had never used a language that made me consider memory in quite the same manner. Yes, I have become used to C/C++ memory management via pointers and allocating/declaring memory on the heap. But Rust presents its own challenges that really took time to understand. I ran this lab exclusively on Codio, so it might be possible that the execution runtime might be improved on a different setup such as my own PC that has much more processing power than Codio (my computer is a gaming PC exclusively). Overall, this lab took around 30 hours, including the 10 hours I spent going through the provided Rust documentation and the 2PC example video I posted at the beginning of this report. The last 5 hours was largely devoted to clearing up any minor issues and putting together this report.