Lab 4: Two-Phase Commit Protocol in Rust

Detailed approach

Qiying Wu

November 20, 2024

Abstract

This report documents the implementation of a simple Two-Phase Commit (2PC) protocol in Rust as part of the CS380P: Parallel Systems course. The project focuses on two primary objectives: leveraging Rust's type system and concurrency model to ensure safe and efficient parallel execution, and gaining practical experience with distributed protocols.

The implementation supports a single coordinator, multiple clients issuing transaction requests, and multiple participants who execute operations and vote on transaction outcomes. Fault-tolerance mechanisms handle participant failures, including operation aborts and silent message losses, while ensuring correctness as verified by log analysis. The project does not address coordinator failure or recovery protocols but includes provisions for participants to rejoin silently after failure.

Performance was evaluated under various configurations, including different numbers of participants, clients, and transaction probabilities. The results highlight the efficiency of Rust's concurrency primitives and the correctness of the 2PC implementation. Challenges encountered during the lab included managing inter-process communication using Rust's 'ipc-channel' crate, adhering to Rust's ownership model, and handling concurrency safely.

This report provides an in-depth discussion of the implementation approach, performance evaluation, and insights gained. Additionally, potential extensions such as implementing recovery protocols and supporting coordinator failure are outlined for future work.

Introduction

The primary goal of this assignment is two-fold. First, it aims to provide practical exposure to Rust, a modern systems programming language that emphasizes memory safety and concurrency guarantees through its innovative type system. This includes leveraging Rust's ownership model, borrowing rules, and concurrency primitives to implement a safe and efficient distributed system. Second, it seeks to familiarize students with the Two-Phase Commit (2PC) protocol, a fundamental distributed algorithm used to achieve consensus in transactional systems. By implementing a simplified version of 2PC, the project enables hands-on experience with distributed protocol design, while focusing on the correctness of the commit/abort decision across multiple processes.

The Two-Phase Commit (2PC) protocol is a widely used algorithm in distributed systems to ensure atomicity in transactions involving multiple participants. It operates in two distinct phases: the *prepare phase*, where a coordinator solicits votes from participants to either commit or abort the transaction, and the *commit phase*, where the coordinator enforces the outcome based on the participants' votes. If all participants vote to commit, the coordinator issues a commit command; otherwise, it instructs all participants to abort.

The protocol is crucial in scenarios where consistency and reliability are required, such as distributed databases, transaction management, and fault-tolerant systems. By implementing the 2PC protocol, this assignment provides insights into the complexities of achieving consensus in the presence of participant failures and unreliable communication, while offering a practical introduction to distributed systems programming. The simplified scope of the assignment excludes coordinator failures and recovery protocols, allowing a focused exploration of participant interactions and state management in a distributed environment.

Approach

0.1 Design Overview

Coordinator

The coordinator is responsible for managing the 2PC protocol. Its tasks include:

- Initializing and maintaining communication channels with clients and participants using Inter-Process Communication (IPC).
- Registering clients and participants via the `client_join` and `participant_join` methods, respectively.
- Coordinating the prepare and commit phases of the protocol through the `start()` method.
- Using threads and synchronization primitives like `Arc`, `Mutex`, and `Barrier` to manage concurrency and shared state.

Client

Clients generate transaction requests and communicate with the coordinator:

- Each client is spawned as a child process using the `spawn_child_and_connect()` function, which sets up IPC channels.
- The client sends requests to the coordinator and waits for commit or abort responses.
- The `run_client()` function handles the client's lifecycle and ensures that requests are executed in a loop until completion.

Participant

Participants execute the operations requested by the coordinator and vote in the prepare phase:

- Each participant is spawned as a child process, similar to clients, with unique IPC channels for communication.
- The `run_participant()` function initializes the participant's log and manages its connection to the coordinator.
- Participants handle success and failure scenarios for both message delivery and operation execution, as specified by the command-line parameters `-s` and `-S`.

Message Communication

To enable inter-process communication in the implementation, we can utilize the `ipc-channel` crate. This crate provides a lightweight mechanism for sending and receiving messages between processes, which is crucial for coordinating the *Two-Phase Commit* (2PC) protocol. Below, I outline the key aspects of message communication in my implementation:

0.2 Implementation

Main Function and System Initialization

The main function orchestrates the overall execution of the Two-Phase Commit (2PC) protocol, managing the lifecycle of the coordinator, clients, and participants. Below are the key components and their implementation details:

- **CLI Argument Parsing and Logging:**
  - Command-line arguments are parsed using the `TPCOptions` struct to configure runtime parameters such as the number of clients, participants, and requests, as well as logging and communication options.
  - Logging is set up using the `stderrlog` crate, enabling detailed runtime diagnostics with adjustable verbosity.
- **Signal Handling:**
  - A `Ctrl+C` handler is registered using the `ctrlc` crate to manage graceful termination.
  - An atomic flag (`running`) is shared across threads and processes to signal the system to shut down cleanly.
- **Mode-Based Execution:**
  - The program operates in one of four modes, determined by the `--mode` CLI parameter:

    * `run`: Executes the full 2PC protocol by initializing the coordinator, clients, and participants.
    * `client`: Launches a client process to issue transaction requests to the coordinator.
    * `participant`: Launches a participant process to receive proposals, vote, and act on global decisions.
    * `check`: Validates the logs of the previous run to ensure correctness.

- **Coordinator Initialization:**
  - The `run()` function initializes the coordinator with persistent logging and thread-safe state management using `Arc<Mutex<>>`.
  - Clients and participants are dynamically registered with the coordinator using separate IPC channels for communication.
  - A synchronization barrier ensures the coordinator is ready before clients and participants proceed.

- **Client and Participant Management:**
  - The `spawn_child_and_connect()` function sets up IPC channels and spawns child processes for clients and participants.
  - Each client and participant process connects back to the coordinator, establishing bidirectional communication channels for transaction processing.

- **Coordinator Execution:**
  - The coordinator runs in a separate thread, managing transaction requests from clients and votes from participants according to the 2PC protocol.
  - Global decisions (`Commit` or `Abort`) are broadcast to all participants and the initiating client.

- **Shutdown and Cleanup:**
  - The main thread waits for all child processes (clients and participants) to complete before joining the coordinator thread.
  - The coordinator reports a summary of committed, aborted, and unknown transactions before termination.

This implementation effectively manages the lifecycle of the 2PC protocol, ensuring robust communication and coordination between processes while supporting clean termination and detailed logging for correctness verification.

Coordinator Implementation

The coordinator acts as the central entity in the Two-Phase Commit (2PC) protocol, responsible for orchestrating transactions between clients and participants. Its implementation includes the following components:

- **Initialization:**
  - The `new()` constructor initializes the coordinator's state, logs, and communication channels. It sets up:
    - A state machine defined by the `CoordinatorState` enum, which includes states such as `Quiescent`, `ProposalSent`, and `ReceivedVotesCommit`.
    - Maps for managing IPC communication channels with clients and participants.
    - A persistent log using the `OpLog` API to record transaction decisions.
- **Joining Clients and Participants:**
  - The `participant_join()` method tracks communication channels for newly added participants.
  - The `client_join()` method ensures new clients are only added when the coordinator is in a `Quiescent` state.
- **Transaction Protocol:**
  - The `protocol()` method implements the coordinator's side of the 2PC protocol and manages the following phases:
    - **Phase 1: Proposal**
      - Waits for client transaction requests while in the `Quiescent` state.
      - Broadcasts a `CoordinatorPropose` message to all participants after receiving a request.
    - **Phase 2: Global Decision**
      - Collects votes from participants in the `ProposalSent` state.
      - If all participants vote `Commit`, sends a global `CoordinatorCommit` decision.
      - If any participant votes `Abort`, or in case of timeout, sends a global `CoordinatorAbort` decision.
      - Broadcasts the final decision to all participants and notifies the initiating client.
    - **State Reset:**
      - After broadcasting the global decision, transitions back to the `Quiescent` state to handle subsequent transactions.
- **Broadcasting Decisions:**

  - The `broadcast_global_decision()` method sends the final transaction decision (`Commit` or `Abort`) to all participants.
  - It converts the decision into a client-specific result message (`ClientResultCommit` or `ClientResultAbort`) and sends it to the initiating client.

- **Timeout Handling:**
  - If participant votes are not received within a fixed duration, the coordinator defaults to an `Abort` decision to maintain consistency.

- **Logging and Status Reporting:**
  - All transaction phases are recorded in the persistent log to ensure durability and traceability.
  - The `report_status()` method summarizes the total number of committed, aborted, and unknown transactions before the coordinator shuts down.

This implementation ensures that the coordinator maintains consistency and reliability while handling concurrent client requests and participant votes. The use of a state machine and persistent logging provides robustness in managing transactions and handling failures.

Client Implementation

The client is responsible for initiating transaction requests to the coordinator and receiving the results of those requests. Its implementation includes the following key aspects:

- **Initialization:**
  - Each client is initialized with a unique identifier, a shared atomic flag (`running`) for managing the protocol's execution, and separate IPC channels for communication with the coordinator.
  - The `new()` constructor initializes the client's state, including counters for successful, failed, and unknown transactions, as well as a map to track the status of each transaction.

- **Protocol Execution:**
  - The `protocol()` method manages the client's lifecycle during the Two-Phase Commit (2PC) protocol. It:
    * Iteratively sends transaction requests up to a specified number (`-r`).
    * Waits for the coordinator's response for each request before issuing the next.
    * Stops early if the simulation ends or the coordinator shuts down.
  - After all requests are processed, the client waits for an exit signal from the `running` flag.

- **Sending Operations:**

  - The `send_next_operation()` method creates a unique transaction ID (TXID) for each request and sends it to the coordinator using the IPC channel.
  - In the event of a failure to send, the transaction status is recorded as `Unknown`, and the client shuts down.

- **Receiving Results:**
  - The `recv_result()` method waits for the coordinator's response for the most recently issued request.
  - Based on the response type:
    - `ClientResultCommit`: The transaction is marked as `Committed`, and the count of successful operations is incremented.
    - `ClientResultAbort`: The transaction is marked as `Aborted`, and the count of failed operations is incremented.
    - `CoordinatorExit`: The client shuts down gracefully.
    - Unexpected message types are logged as errors and marked as `Unknown`.

- **Reporting and Shutdown:**
  - The `report_status()` method summarizes the outcomes of all transactions, providing counts of committed, aborted, and unknown requests.
  - The `shutdown()` method sets the `running` flag to `false` and ensures a clean exit.

- **Exit Handling:**
  - The `wait_for_exit_signal()` method ensures the client remains idle until a termination signal is received. This guarantees that the client does not terminate prematurely while waiting for the coordinator's shutdown signal.

The client design ensures robust communication with the coordinator while adhering to the constraints of the Two-Phase Commit protocol. It handles both normal and failure scenarios effectively, allowing for consistent and reliable transaction processing.

Participant implementation

Participants execute the operations requested by the coordinator and vote in the prepare phase, as part of the Two-Phase Commit (2PC) protocol. Their design and functionality include the following:

- Each participant is spawned as a child process, similar to clients, with unique IPC channels for communication with the coordinator. This ensures independent communication paths and isolation.

- The `run_participant()` function initializes the participant by:
  - Setting up the log for recording commit/abort decisions.

  - Connecting to the coordinator via IPC channels for receiving and sending messages.
  - Initializing runtime parameters such as `-s` (operation success probability) and `-S` (message delivery success probability) as per the command-line inputs.

- Participants implement a state machine defined by the `ParticipantState` enum:
  - `Quiescent`: The participant is idle and waiting for the next proposal.
  - `ReceivedP1`: The participant has received a proposal and is evaluating its operation.
  - `VotedCommit`: The participant voted to commit in Phase 1.
  - `VotedAbort`: The participant voted to abort in Phase 1.
  - `AwaitingGlobalDecision`: The participant is waiting for the global decision in Phase 2.

- The `send()` method handles message delivery to the coordinator, considering the message success probability (`-S`). If delivery fails, it logs the failure and continues execution based on the protocol's state.

- The `perform_operation()` method simulates operation execution with a success or failure outcome determined probabilistically (`-s`). Based on this outcome, the participant sends a vote to the coordinator (`ParticipantVoteCommit` or `ParticipantVoteAbort`).

- Participants handle various message types from the coordinator:
  - `CoordinatorPropose`: Initiates the prepare phase; participants evaluate and vote.
  - `CoordinatorCommit` and `CoordinatorAbort`: Global decisions in Phase 2 are logged and acted upon.
  - `CoordinatorExit`: Signals termination, allowing participants to gracefully shut down.

- The participant maintains a log of all transactions using the `OpLog` API for persistent record-keeping of commit/abort decisions.

- Before termination, participants call `report_status()` to summarize the results of all handled transactions, including counts of committed, aborted, and unknown states.

- Participants continuously monitor an exit signal via an atomic flag (`running`) to ensure proper cleanup and termination when the protocol concludes.

Concurrency and Synchronization

Rust's `Arc`, `Mutex`, and atomic variables are used extensively to manage shared state between threads. A `Barrier` is employed to synchronize the initialization of the coordinator and its child processes. The program ensures safety and concurrency correctness through Rust's ownership and type system.

Concurrency Mechanisms in Rust

To manage concurrency and ensure thread safety, various Rust primitives were utilized effectively:

- `Arc` **(Atomic Reference Counting):** Enabled shared ownership of immutable and thread-safe references, such as the coordinator instance and the running state, across threads.
- `Mutex` **(Mutual Exclusion):** Ensured safe access and modification of the coordinators internal state, such as mappings of participants and clients. The `Coordinator` was wrapped in an `Arc<Mutex<...>>`, allowing multiple threads to interact with it safely.
- `AtomicBool`**:** Managed the `running` state atomically to allow clean shutdown handling when a SIGINT (Ctrl+C) was received, avoiding race conditions.
- `Barrier`**:** Synchronized initialization between the main thread and the coordinator thread, ensuring all components were ready before transaction processing began.
- `std::thread`**:** Facilitated the creation of separate threads for the coordinators protocol execution and child processes (clients and participants). Each thread operated independently, with synchronization handled through shared primitives and channels.

Inter-Process Communication and Protocol Logic

The system leveraged non-blocking communication mechanisms for efficient message handling:

- `ipc-channel` **library:** Enabled type-safe and efficient bidirectional communication via `IpcSender` and `IpcReceiver`.
- **Non-blocking message polling:** The `Coordinator` used `try_recv()` to poll for messages from participants and clients without blocking indefinitely.
- **State synchronization:** The coordinator managed protocol states (e.g., `ReceivedRequest`, `ProposalSent`, `ReceivedVotesCommit`) and transitions while ensuring consistent updates to shared state using locks and atomic variables.
- **Timeout handling:** The protocol incorporated timeout mechanisms to handle delayed or missing responses from participants, ensuring robustness.
- **Global decisions:** The coordinator broadcasted commit/abort decisions to participants and the initiating client, ensuring the consistency of transaction results across all components.

Each message type is serialized and deserialized using the `ipc-channel`'s built-in mechanisms, ensuring that the message format is consistent and type-safe across all processes.

Execution Modes

The program supports the following modes:

- **Run Mode:** Launches the coordinator, spawns client and participant processes, and executes the 2PC protocol.

  ```
  ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m run
  ./target/debug/two_phase_commit -s .95 -S .95 -c 4 -p 10 -r 10 -m run
  ```

  This command runs the Two-Phase Commit (2PC) protocol with the specified configuration options:

  - `-s .95`: Sets the probability that a participant will successfully send a message during the 2PC protocol to 95%.
  - `-S .95`: Sets the probability that a participant will successfully execute an operation (commit or abort) during the 2PC protocol to 95%. This simulates operational reliability in addition to communication reliability.
  - `-c 4`: Specifies that 4 client processes will be spawned. Each client sends transaction requests to the coordinator.
  - `-p 10`: Indicates that 10 participant processes will handle voting and global decision phases of the protocol.
  - `-r 10`: Configures each client to issue 10 transaction requests during the execution.
  - `-m run`: Sets the program mode to `run`, which initializes the coordinator, spawns client and participant processes, and executes the 2PC protocol.

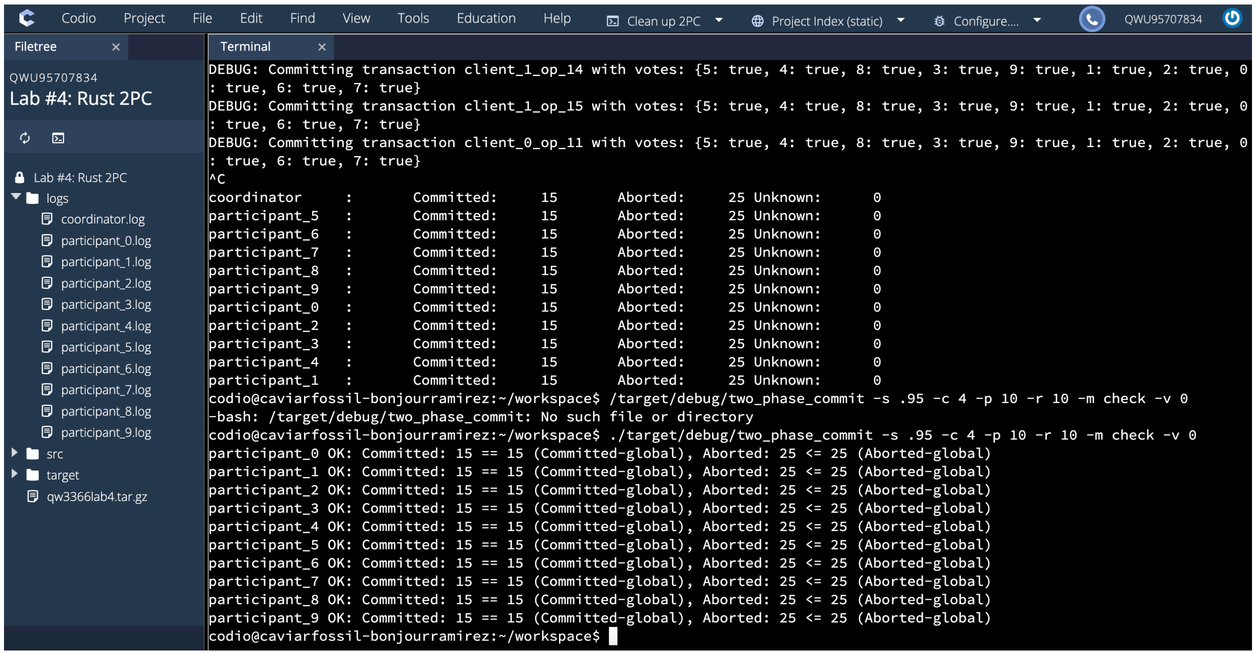
- **Client/Participant Modes:** Dedicated to the execution of client and participant processes.

- **Check Mode:** Verifies the correctness of the protocol using log files produced during the run.

  ```
  ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m check -v 0
  ```

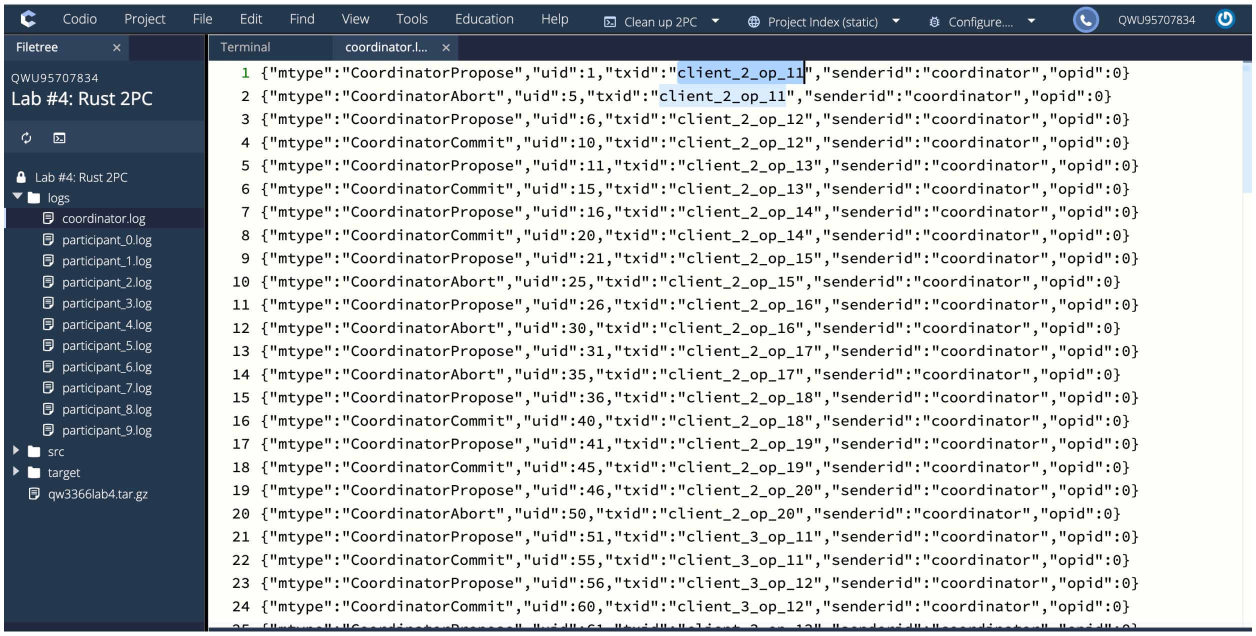
Result

Correctness



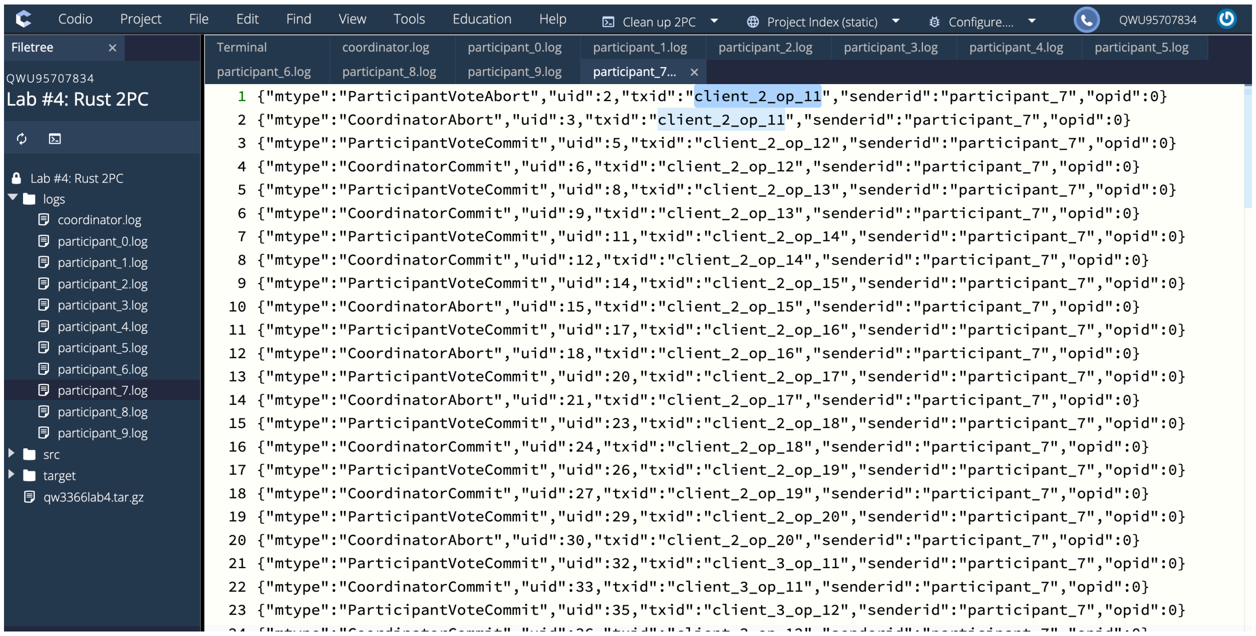
```
DEBUG: Committing transaction client_1_op_14 with votes: {5: true, 4: true, 8: true, 3: true, 9: true, 1: true, 2: true, 0
: true, 6: true, 7: true}
DEBUG: Committing transaction client_1_op_15 with votes: {5: true, 4: true, 8: true, 3: true, 9: true, 1: true, 2: true, 0
: true, 6: true, 7: true}
DEBUG: Committing transaction client_0_op_11 with votes: {5: true, 4: true, 8: true, 3: true, 9: true, 1: true, 2: true, 0
: true, 6: true, 7: true}
^C
coordinator     :       Committed:     15       Aborted:     25 Unknown:      0
participant_5   :       Committed:     15       Aborted:     25 Unknown:      0
participant_6   :       Committed:     15       Aborted:     25 Unknown:      0
participant_7   :       Committed:     15       Aborted:     25 Unknown:      0
participant_8   :       Committed:     15       Aborted:     25 Unknown:      0
participant_9   :       Committed:     15       Aborted:     25 Unknown:      0
participant_0   :       Committed:     15       Aborted:     25 Unknown:      0
participant_2   :       Committed:     15       Aborted:     25 Unknown:      0
participant_3   :       Committed:     15       Aborted:     25 Unknown:      0
participant_4   :       Committed:     15       Aborted:     25 Unknown:      0
participant_1   :       Committed:     15       Aborted:     25 Unknown:      0
codio@caviarfossil-bonjourramirez:~/workspace$ /target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m check -v 0
-bash: /target/debug/two_phase_commit: No such file or directory
codio@caviarfossil-bonjourramirez:~/workspace$ ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m check -v 0
participant_0 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_1 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_2 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_3 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_4 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_5 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_6 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_7 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_8 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
participant_9 OK: Committed: 15 == 15 (Committed-global), Aborted: 25 <= 25 (Aborted-global)
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 1: Correctness Verification: Protocol Execution Output



```
{"mtype":"CoordinatorPropose","uid":1,"txid":"client_2_op_11","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorAbort","uid":5,"txid":"client_2_op_11","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":6,"txid":"client_2_op_12","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":10,"txid":"client_2_op_12","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":11,"txid":"client_2_op_13","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":15,"txid":"client_2_op_13","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":16,"txid":"client_2_op_14","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":20,"txid":"client_2_op_14","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":21,"txid":"client_2_op_15","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorAbort","uid":25,"txid":"client_2_op_15","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":26,"txid":"client_2_op_16","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorAbort","uid":30,"txid":"client_2_op_16","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":31,"txid":"client_2_op_17","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorAbort","uid":35,"txid":"client_2_op_17","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":36,"txid":"client_2_op_18","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":40,"txid":"client_2_op_18","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":41,"txid":"client_2_op_19","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":45,"txid":"client_2_op_19","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":46,"txid":"client_2_op_20","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorAbort","uid":50,"txid":"client_2_op_20","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":51,"txid":"client_3_op_11","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":55,"txid":"client_3_op_11","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorPropose","uid":56,"txid":"client_3_op_12","senderid":"coordinator","opid":0}
{"mtype":"CoordinatorCommit","uid":60,"txid":"client_3_op_12","senderid":"coordinator","opid":0}
```

Figure 2: Correctness Verification: coordinator log



```
{"mtype":"ParticipantVoteAbort","uid":2,"txid":"client_2_op_11","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorAbort","uid":3,"txid":"client_2_op_11","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":5,"txid":"client_2_op_12","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":6,"txid":"client_2_op_12","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":8,"txid":"client_2_op_13","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":9,"txid":"client_2_op_13","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":11,"txid":"client_2_op_14","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":12,"txid":"client_2_op_14","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":14,"txid":"client_2_op_15","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorAbort","uid":15,"txid":"client_2_op_15","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":17,"txid":"client_2_op_16","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorAbort","uid":18,"txid":"client_2_op_16","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":20,"txid":"client_2_op_17","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorAbort","uid":21,"txid":"client_2_op_17","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":23,"txid":"client_2_op_18","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":24,"txid":"client_2_op_18","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":26,"txid":"client_2_op_19","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":27,"txid":"client_2_op_19","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":29,"txid":"client_2_op_20","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorAbort","uid":30,"txid":"client_2_op_20","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":32,"txid":"client_3_op_11","senderid":"participant_7","opid":0}
{"mtype":"CoordinatorCommit","uid":33,"txid":"client_3_op_11","senderid":"participant_7","opid":0}
{"mtype":"ParticipantVoteCommit","uid":35,"txid":"client_3_op_12","senderid":"participant_7","opid":0}
```

Figure 3: Correctness Verification: participant log

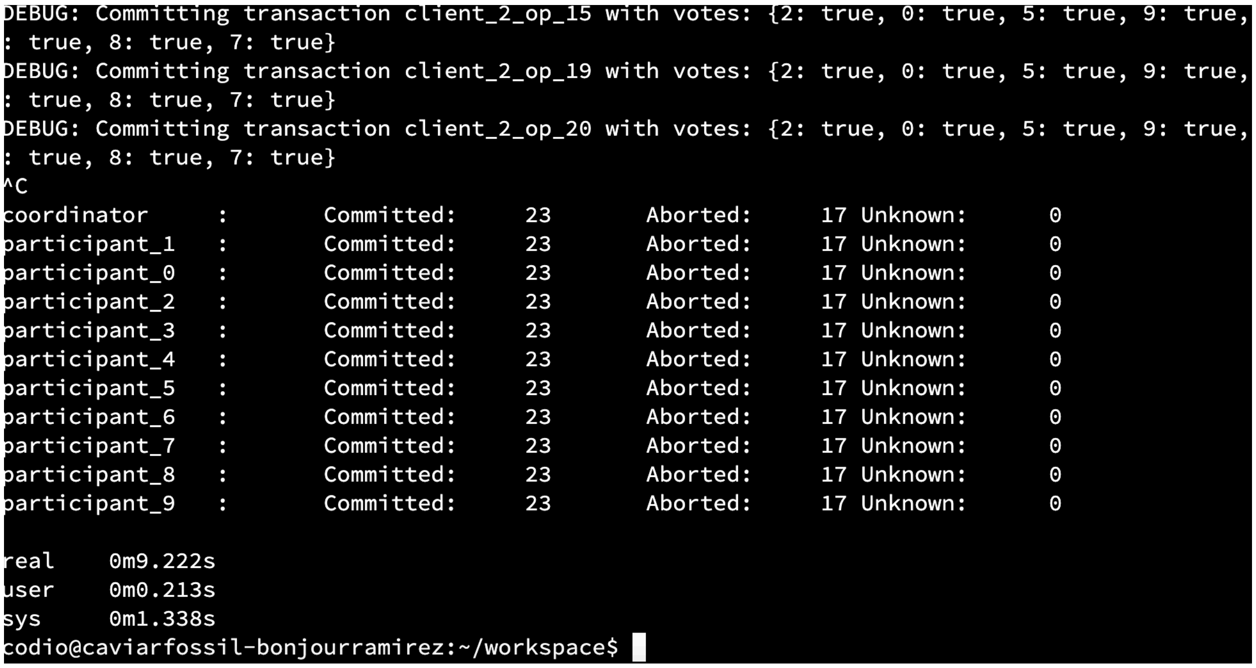
Checking the log in a close look, we can see that for client_2_op_11, participant_7 voted to abort, and coordinator aborted.
These screenshots confirm that the protocol adheres to the following correctness properties:

- **Atomicity:** Transactions are either fully committed or fully aborted.
- **Consistency:** The global decision (commit or abort) aligns with the votes received from all participants.
- **Durability:** Decisions are logged persistently, ensuring recovery in the event of failures.

Performance Evaluation

Running the following command and we get the results as the picture below.

```
time ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m run
```



```
DEBUG: Committing transaction client_2_op_15 with votes: {2: true, 0: true, 5: true, 9: true,
: true, 8: true, 7: true}
DEBUG: Committing transaction client_2_op_19 with votes: {2: true, 0: true, 5: true, 9: true,
: true, 8: true, 7: true}
DEBUG: Committing transaction client_2_op_20 with votes: {2: true, 0: true, 5: true, 9: true,
: true, 8: true, 7: true}
^C
coordinator     :       Committed:     23       Aborted:     17 Unknown:      0
participant_1   :       Committed:     23       Aborted:     17 Unknown:      0
participant_0   :       Committed:     23       Aborted:     17 Unknown:      0
participant_2   :       Committed:     23       Aborted:     17 Unknown:      0
participant_3   :       Committed:     23       Aborted:     17 Unknown:      0
participant_4   :       Committed:     23       Aborted:     17 Unknown:      0
participant_5   :       Committed:     23       Aborted:     17 Unknown:      0
participant_6   :       Committed:     23       Aborted:     17 Unknown:      0
participant_7   :       Committed:     23       Aborted:     17 Unknown:      0
participant_8   :       Committed:     23       Aborted:     17 Unknown:      0
participant_9   :       Committed:     23       Aborted:     17 Unknown:      0

real    0m9.222s
user    0m0.213s
sys     0m1.338s
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 4: Execution Time: execution time

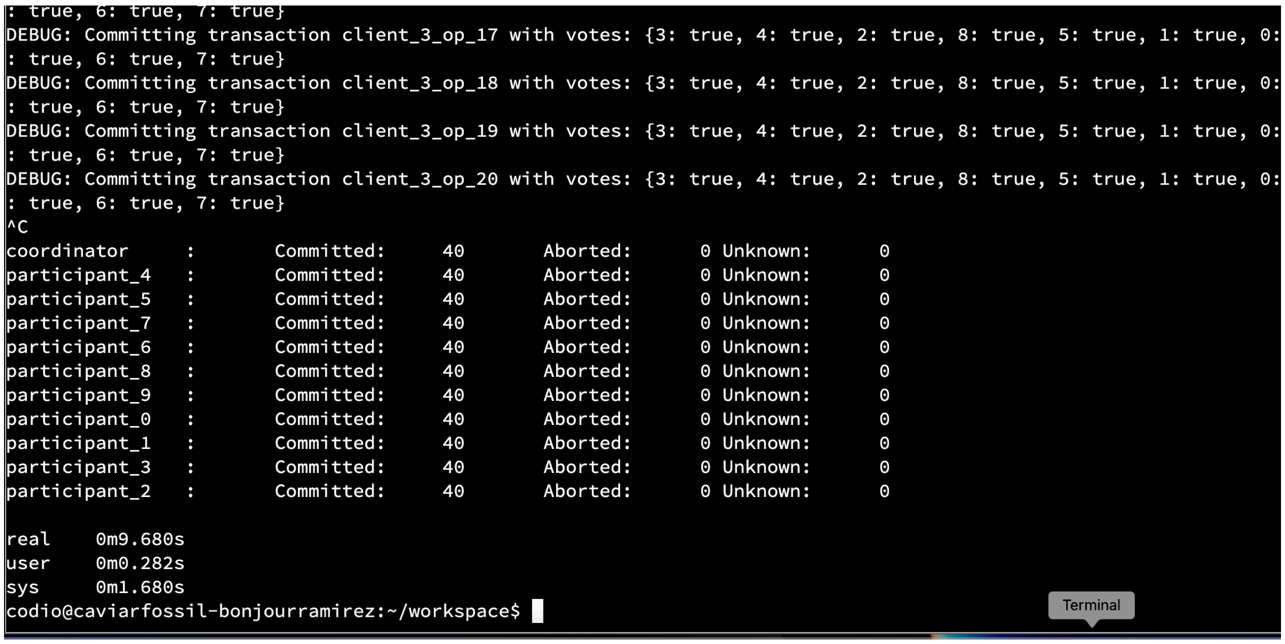
The `time` command measures the program's execution time, providing three key metrics:

- **`real` (e.g., 9.222 seconds):** The total elapsed wall-clock time from when the program started until it was interrupted by `Ctrl+C`. This includes both active processing time and any waiting or idle time (e.g., for inter-process communication (IPC) or participant responses).
- **`user` (e.g., 0.213 seconds):** The total CPU time spent executing the program's instructions in user mode (i.e., the actual processing performed by my code).
- **`sys` (e.g., 1.338 seconds):** The total CPU time spent on system calls in kernel mode (e.g., file I/O operations, thread synchronization, or IPC).

Execution time comparison

No let's compare the execution time for different number of clients, requests and participants, with a baseline, the picture below is one of the run result for the command provided in instructions.

```
time ./target/debug/two_phase_commit -c 4 -p 10 -r 10 -m run
```

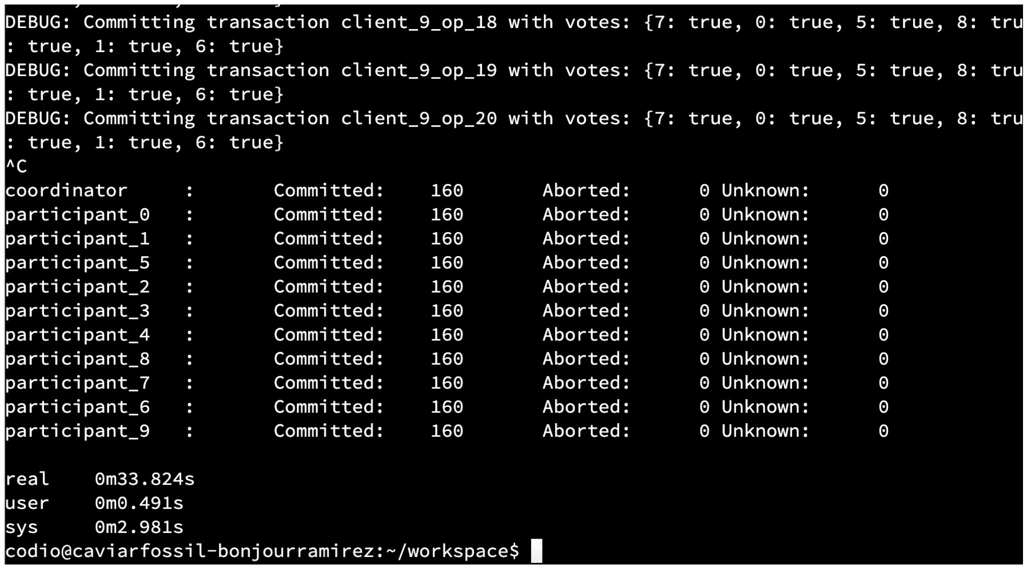


```
: true, 6: true, 7: true}
DEBUG: Committing transaction client_3_op_17 with votes: {3: true, 4: true, 2: true, 8: true, 5: true, 1: true, 0:
: true, 6: true, 7: true}
DEBUG: Committing transaction client_3_op_18 with votes: {3: true, 4: true, 2: true, 8: true, 5: true, 1: true, 0:
: true, 6: true, 7: true}
DEBUG: Committing transaction client_3_op_19 with votes: {3: true, 4: true, 2: true, 8: true, 5: true, 1: true, 0:
: true, 6: true, 7: true}
DEBUG: Committing transaction client_3_op_20 with votes: {3: true, 4: true, 2: true, 8: true, 5: true, 1: true, 0:
: true, 6: true, 7: true}
^C
coordinator     :       Committed:     40       Aborted:      0 Unknown:      0
participant_4   :       Committed:     40       Aborted:      0 Unknown:      0
participant_5   :       Committed:     40       Aborted:      0 Unknown:      0
participant_7   :       Committed:     40       Aborted:      0 Unknown:      0
participant_6   :       Committed:     40       Aborted:      0 Unknown:      0
participant_8   :       Committed:     40       Aborted:      0 Unknown:      0
participant_9   :       Committed:     40       Aborted:      0 Unknown:      0
participant_0   :       Committed:     40       Aborted:      0 Unknown:      0
participant_1   :       Committed:     40       Aborted:      0 Unknown:      0
participant_3   :       Committed:     40       Aborted:      0 Unknown:      0
participant_2   :       Committed:     40       Aborted:      0 Unknown:      0

real    0m9.680s
user    0m0.282s
sys     0m1.680s
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 5: Execution Time: without s flag, 4 clients

```
time ./target/debug/two_phase_commit -c 16 -p 10 -r 10 -m run
```

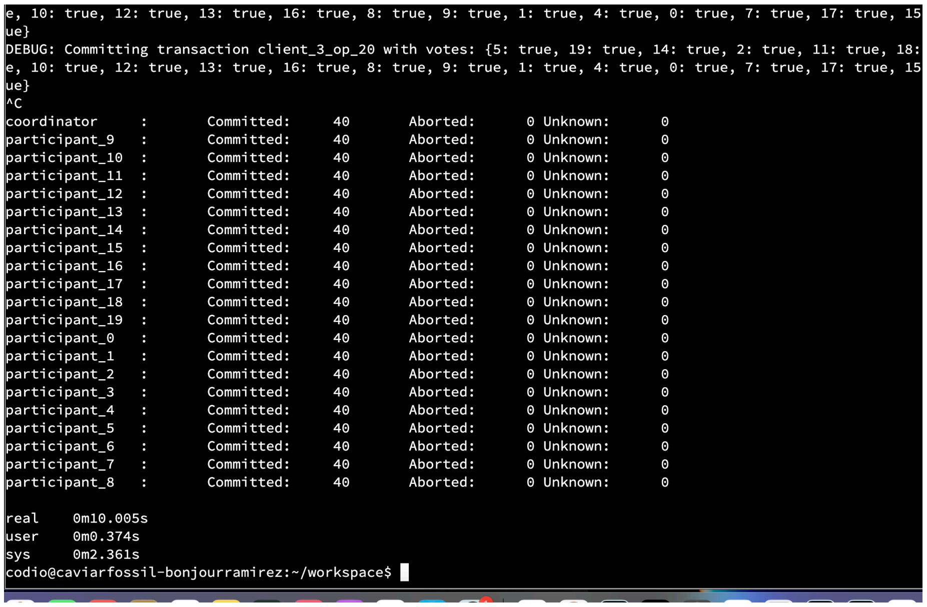


```
DEBUG: Committing transaction client_9_op_18 with votes: {7: true, 0: true, 5: true, 8: tru
: true, 1: true, 6: true}
DEBUG: Committing transaction client_9_op_19 with votes: {7: true, 0: true, 5: true, 8: tru
: true, 1: true, 6: true}
DEBUG: Committing transaction client_9_op_20 with votes: {7: true, 0: true, 5: true, 8: tru
: true, 1: true, 6: true}
^C
coordinator     :       Committed:    160       Aborted:      0 Unknown:      0
participant_0   :       Committed:    160       Aborted:      0 Unknown:      0
participant_1   :       Committed:    160       Aborted:      0 Unknown:      0
participant_5   :       Committed:    160       Aborted:      0 Unknown:      0
participant_2   :       Committed:    160       Aborted:      0 Unknown:      0
participant_3   :       Committed:    160       Aborted:      0 Unknown:      0
participant_4   :       Committed:    160       Aborted:      0 Unknown:      0
participant_8   :       Committed:    160       Aborted:      0 Unknown:      0
participant_7   :       Committed:    160       Aborted:      0 Unknown:      0
participant_6   :       Committed:    160       Aborted:      0 Unknown:      0
participant_9   :       Committed:    160       Aborted:      0 Unknown:      0

real    0m33.824s
user    0m0.491s
sys     0m2.981s
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 6: Execution Time: without s flag, 16 clients

```
time ./target/debug/two_phase_commit -c 4 -p 20 -r 10 -m run
```

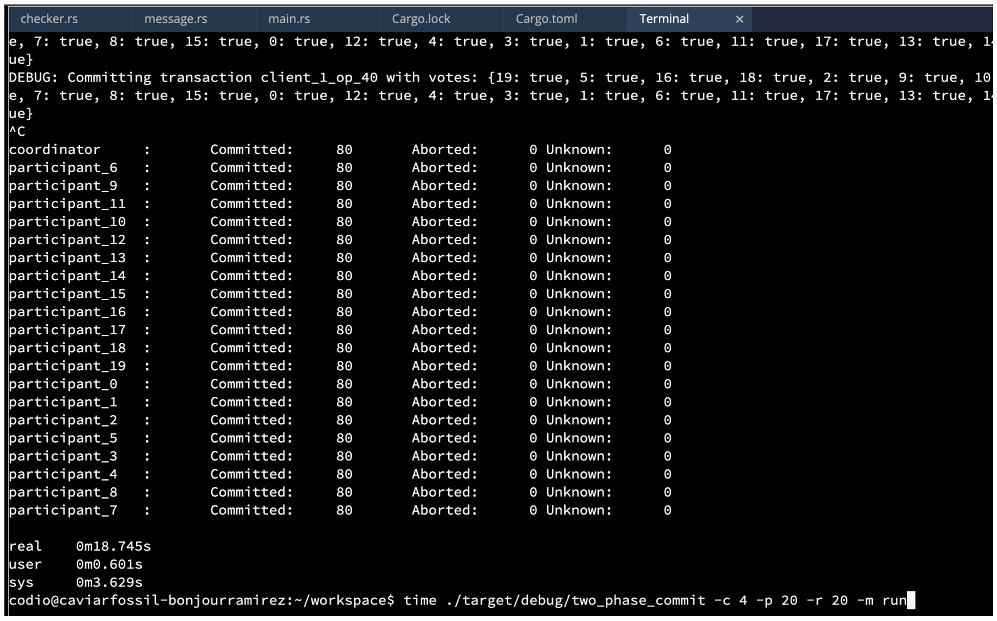


```
e, 10: true, 12: true, 13: true, 16: true, 8: true, 9: true, 1: true, 4: true, 0: true, 7: true, 17: true, 15
ue}
DEBUG: Committing transaction client_3_op_20 with votes: {5: true, 19: true, 14: true, 2: true, 11: true, 18:
e, 10: true, 12: true, 13: true, 16: true, 8: true, 9: true, 1: true, 4: true, 0: true, 7: true, 17: true, 15
ue}
^C
coordinator     :       Committed:     40       Aborted:      0 Unknown:      0
participant_9   :       Committed:     40       Aborted:      0 Unknown:      0
participant_10  :       Committed:     40       Aborted:      0 Unknown:      0
participant_11  :       Committed:     40       Aborted:      0 Unknown:      0
participant_12  :       Committed:     40       Aborted:      0 Unknown:      0
participant_13  :       Committed:     40       Aborted:      0 Unknown:      0
participant_14  :       Committed:     40       Aborted:      0 Unknown:      0
participant_15  :       Committed:     40       Aborted:      0 Unknown:      0
participant_16  :       Committed:     40       Aborted:      0 Unknown:      0
participant_17  :       Committed:     40       Aborted:      0 Unknown:      0
participant_18  :       Committed:     40       Aborted:      0 Unknown:      0
participant_19  :       Committed:     40       Aborted:      0 Unknown:      0
participant_0   :       Committed:     40       Aborted:      0 Unknown:      0
participant_1   :       Committed:     40       Aborted:      0 Unknown:      0
participant_2   :       Committed:     40       Aborted:      0 Unknown:      0
participant_3   :       Committed:     40       Aborted:      0 Unknown:      0
participant_4   :       Committed:     40       Aborted:      0 Unknown:      0
participant_5   :       Committed:     40       Aborted:      0 Unknown:      0
participant_6   :       Committed:     40       Aborted:      0 Unknown:      0
participant_7   :       Committed:     40       Aborted:      0 Unknown:      0
participant_8   :       Committed:     40       Aborted:      0 Unknown:      0

real    0m10.005s
user    0m0.374s
sys     0m2.361s
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 7: Execution Time: without s flag, 20 participants

```
time ./target/debug/two_phase_commit -c 4 -p 20 -r 20 -m run
```

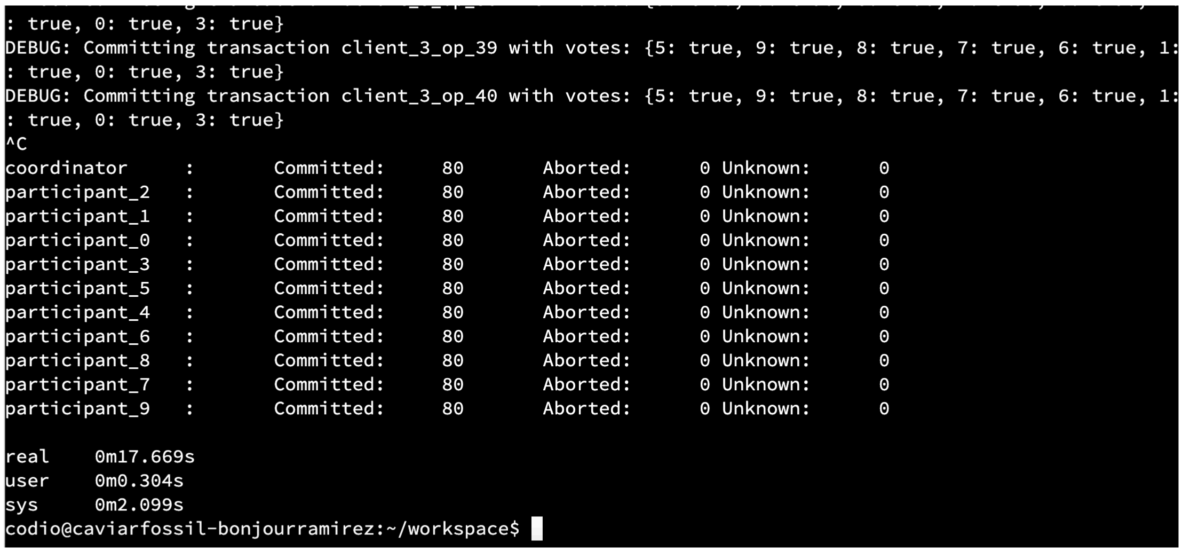


```
e, 7: true, 8: true, 15: true, 0: true, 12: true, 4: true, 3: true, 1: true, 6: true, 11: true, 17: true, 13: true, 1
ue}
DEBUG: Committing transaction client_1_op_40 with votes: {19: true, 5: true, 16: true, 18: true, 2: true, 9: true, 10
e, 7: true, 8: true, 15: true, 0: true, 12: true, 4: true, 3: true, 1: true, 6: true, 11: true, 17: true, 13: true, 1
ue}
^C
coordinator     :       Committed:     80       Aborted:      0 Unknown:      0
participant_6   :       Committed:     80       Aborted:      0 Unknown:      0
participant_9   :       Committed:     80       Aborted:      0 Unknown:      0
participant_11  :       Committed:     80       Aborted:      0 Unknown:      0
participant_10  :       Committed:     80       Aborted:      0 Unknown:      0
participant_12  :       Committed:     80       Aborted:      0 Unknown:      0
participant_13  :       Committed:     80       Aborted:      0 Unknown:      0
participant_14  :       Committed:     80       Aborted:      0 Unknown:      0
participant_15  :       Committed:     80       Aborted:      0 Unknown:      0
participant_16  :       Committed:     80       Aborted:      0 Unknown:      0
participant_17  :       Committed:     80       Aborted:      0 Unknown:      0
participant_18  :       Committed:     80       Aborted:      0 Unknown:      0
participant_19  :       Committed:     80       Aborted:      0 Unknown:      0
participant_0   :       Committed:     80       Aborted:      0 Unknown:      0
participant_1   :       Committed:     80       Aborted:      0 Unknown:      0
participant_2   :       Committed:     80       Aborted:      0 Unknown:      0
participant_5   :       Committed:     80       Aborted:      0 Unknown:      0
participant_3   :       Committed:     80       Aborted:      0 Unknown:      0
participant_4   :       Committed:     80       Aborted:      0 Unknown:      0
participant_8   :       Committed:     80       Aborted:      0 Unknown:      0
participant_7   :       Committed:     80       Aborted:      0 Unknown:      0

real    0m18.745s
user    0m0.601s
sys     0m3.629s
codio@caviarfossil-bonjourramirez:~/workspace$ time ./target/debug/two_phase_commit -c 4 -p 20 -r 20 -m run
```

Figure 8: Execution Time: without s flag, 20 participants, 20 transactions requests

```
time ./target/debug/two_phase_commit -c 4 -p 10 -r 20 -m run
```



```
: true, 0: true, 3: true}
DEBUG: Committing transaction client_3_op_39 with votes: {5: true, 9: true, 8: true, 7: true, 6: true, 1:
: true, 0: true, 3: true}
DEBUG: Committing transaction client_3_op_40 with votes: {5: true, 9: true, 8: true, 7: true, 6: true, 1:
: true, 0: true, 3: true}
^C
coordinator     :       Committed:     80      Aborted:      0 Unknown:      0
participant_2   :       Committed:     80      Aborted:      0 Unknown:      0
participant_1   :       Committed:     80      Aborted:      0 Unknown:      0
participant_0   :       Committed:     80      Aborted:      0 Unknown:      0
participant_3   :       Committed:     80      Aborted:      0 Unknown:      0
participant_5   :       Committed:     80      Aborted:      0 Unknown:      0
participant_4   :       Committed:     80      Aborted:      0 Unknown:      0
participant_6   :       Committed:     80      Aborted:      0 Unknown:      0
participant_8   :       Committed:     80      Aborted:      0 Unknown:      0
participant_7   :       Committed:     80      Aborted:      0 Unknown:      0
participant_9   :       Committed:     80      Aborted:      0 Unknown:      0

real    0m17.669s
user    0m0.304s
sys     0m2.099s
codio@caviarfossil-bonjourramirez:~/workspace$
```

Figure 9: Execution Time: without s flag, 10 participants, 20 transactions

Analyzing the performance of the `two_phase_commit` program under various configurations, based on the `time` command's output:

- **Baseline Configuration** (`-c 4 -p 10 -r 10`)**:**
  - `real = 9.680s`: Total wall-clock time for the program execution.
  - `user = 0.282s`: Minimal CPU time spent on computation.
  - `sys = 1.680s`: Significant time spent on system calls, reflecting the role of IPC and synchronization in the workload.
- **Doubling Requests** (`-c 4 -p 10 -r 20`)**:**
  - `real = 17.669s`: The program execution time nearly doubles due to increased workload from additional transactions.
  - `user = 0.304s`: Minimal increase in computation time, indicating that the workload is not computation-heavy.
  - `sys = 2.099s`: Slight increase in system calls, reflecting the additional IPC messages required for 20 requests.
- **Increasing Participants** (`-c 4 -p 20 -r 20`)**:**
  - `real = 18.745s`: Execution time increases slightly compared to the previous case due to higher communication overhead with more participants.
  - `user = 0.601s`: Higher computation time, as the coordinator handles more participant responses.

  - `sys = 3.629s`: Significant increase in system calls due to additional IPC and synchronization with 20 participants.

- **More Participants with Fewer Requests (`-c 4 -p 20 -r 10`):**
  - `real = 10.005s`: Execution time is slightly longer than the baseline, reflecting the increased number of participants.
  - `user = 0.374s`: Modest increase in computation time to handle additional participants.
  - `sys = 2.361s`: Increased IPC-related system time due to the larger participant pool.

- **Scaling Clients (`-c 16 -p 10 -r 10`):**
  - `real = 33.824s`: Execution time more than triples due to contention among clients for the coordinator's resources.
  - `user = 0.491s`: Slight increase in computation time.
  - `sys = 2.981s`: Increased system time, reflecting the higher number of IPC messages and synchronization.

Key Observations

- **Scaling Requests:** Doubling the number of requests significantly increases wall-clock time (`real`), but computation time (`user`) remains minimal, indicating that the workload is communication-bound.
- **Scaling Participants:** Adding more participants increases both `real` and `sys` times due to higher IPC and synchronization overhead, but `user` time remains modest.
- **Scaling Clients:** Increasing the number of clients leads to a drastic rise in `real` time, likely due to contention at the coordinator. This suggests that the coordinator could be a bottleneck in handling many simultaneous clients.

Conclusion

The program's performance is constrained primarily by communication overhead (reflected in `sys` time) rather than computation (reflected in `user` time). Optimizing IPC and improving the coordinator's ability to handle concurrent clients and participants may significantly enhance scalability.

Insights and Challenges

The project demonstrated the scalability and correctness of the Two-Phase Commit protocol, with predictable increases in execution time under higher client, participant, and request loads. The primary bottleneck was the centralized coordinator, which struggled with high client contention, leading to significant communication overhead (`sys` time). Although the implementation ensured consistency and atomicity, scaling IPC efficiently and managing concurrency posed challenges. Additionally, handling fault tolerance and optimizing for larger distributed systems remain open areas for improvement. The unfamiliarity with Rust's ownership model and concurrency primitives (e.g., `Arc`, `Mutex`, and `AtomicBool`) added to the learning curve but ultimately provided robust safety guarantees in a complex distributed setting.

Time Spent

I spent around 50 hours in total for this lab. Including 10 hours for learning Rust basics, 38 hours for coding, and 2 hours for reprot.