**Srinivas Prasad Prabhu**

**UT EID - sp55629**

**CS380P – Lab 4 – Rust 2 PC**

## Platform Details

CPU - AMD EPYC 7313P 32-Core Processor
RAM - 128 GB RAM
OS - Ubuntu 22.04.3 LTS

rustProj/2pc_rustProj$ rustc --version

**rustc 1.73.0 (cc66ad468 2023-10-03)**

rustProj/2pc_rustProj$ cargo --version

**cargo 1.73.0 (9c4383fb5 2023-08-26)**

## High level approach and description of code implementation

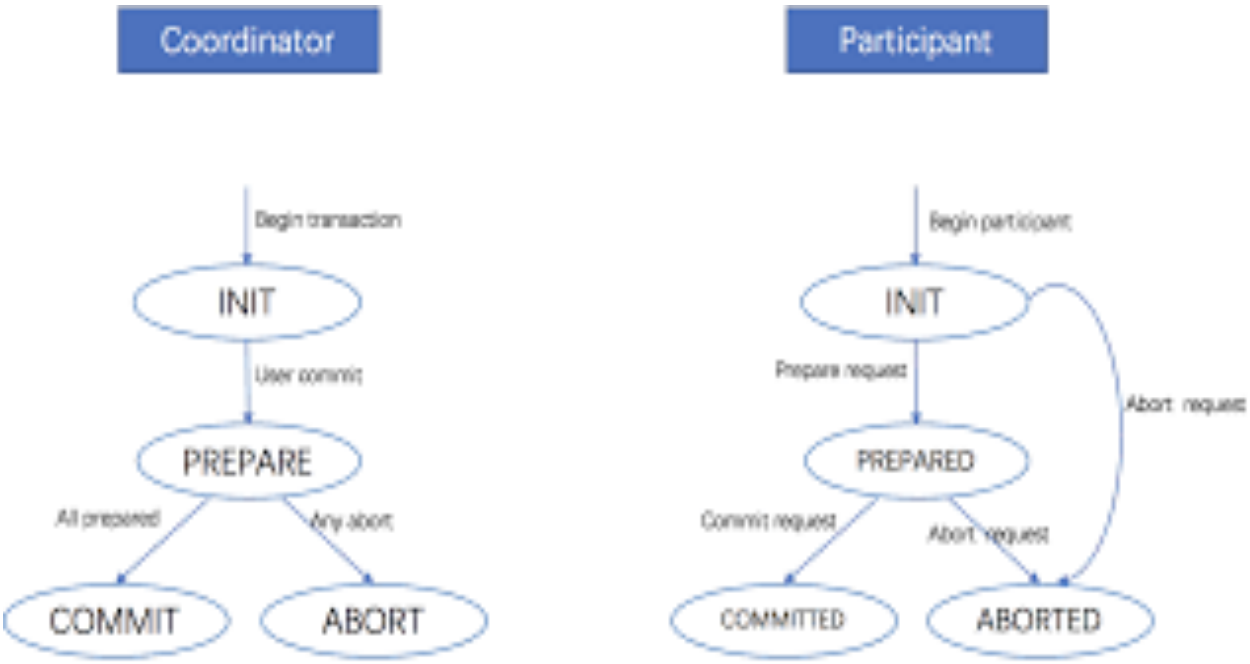
The main program serves as the coordinator. The main program spawns the clients and the participants. Each of these entities communicate with each other using rust ipc channels.

Clients initiate the transactions. The coordinator queues the client requests and serves them in FIFO order.

Coordinator runs the 2-phase commit protocol with all the participants and decides whether to commit or abort the transaction.

This decision is notified to the participants as well as the clients from the coordinator.

The diagram below indicates the state machine on the coordinator and the participant.




**Client**

The client initiates by sending transaction requests to the coordinator. Client sends a request and waits to receive a response from the coordinator before posting its next request. Clients send up to num_requests.

**Coordinator**

The coordinator has 2 parts.

Client facing – The coordinator receives the request and queues them in the msg_q. It then processes each message in the queue in FIFO order.

Once the decision is made, it is communicated to the participant and the client.

Participant facing – Here the coordinator runs the 2-phase commit protocol.

Here it starts off with the coordinator sending the commit prepare/propose message to the participants. The participants can respond with their votes to commit/abort.

If all the participants respond with commit, then the transaction is committed, and the participants and clients are notified to commit.

If any participant responds with abort or there is a timeout, the coordinator decides to abort the transaction and the same is notified to the participants and client.

Transaction status is logged in the coordinator.

## Participant

The participant waits for the coordinator to initiate the 2-phase protocol. On receiving a prepare/propose message from the coordinator, the participant responds with a vote to commit/abort the transaction. It then waits for the coordinator's decision. The coordinator then informs its decision to the participant. The participant follows the coordinator message. Participants response to propose and the final transaction status are logged in the participant.

## Performance gain of code implementation

2-phase commit is a standard protocol that guarantees ACID (atomicity, consistency, isolation, durability) properties of a transaction. However, many modern distributed systems are classified as BASE (basically available, soft-state, eventual consistency). 2-phase commit is generally overkill is any production scenario. 2-phase commit suffers from some important performance pitfalls –

Latency – The coordinator must wait for all the participants to respond or timeout before making any decision.

Single Point of Failure – The coordinator becomes the single point of failure. If the coordinator fails, the entire transaction must be abandoned.

Participant dependency – A single slow participant can make the entire transaction very slow.

Writing log – Since so many logs must be saved, the time for the transaction is also determined by the latency to write and save the logs on the device.

There are 2 main variables that can be varied.

-s – Commit/Abort probability and

-S – Participant vote send probability.

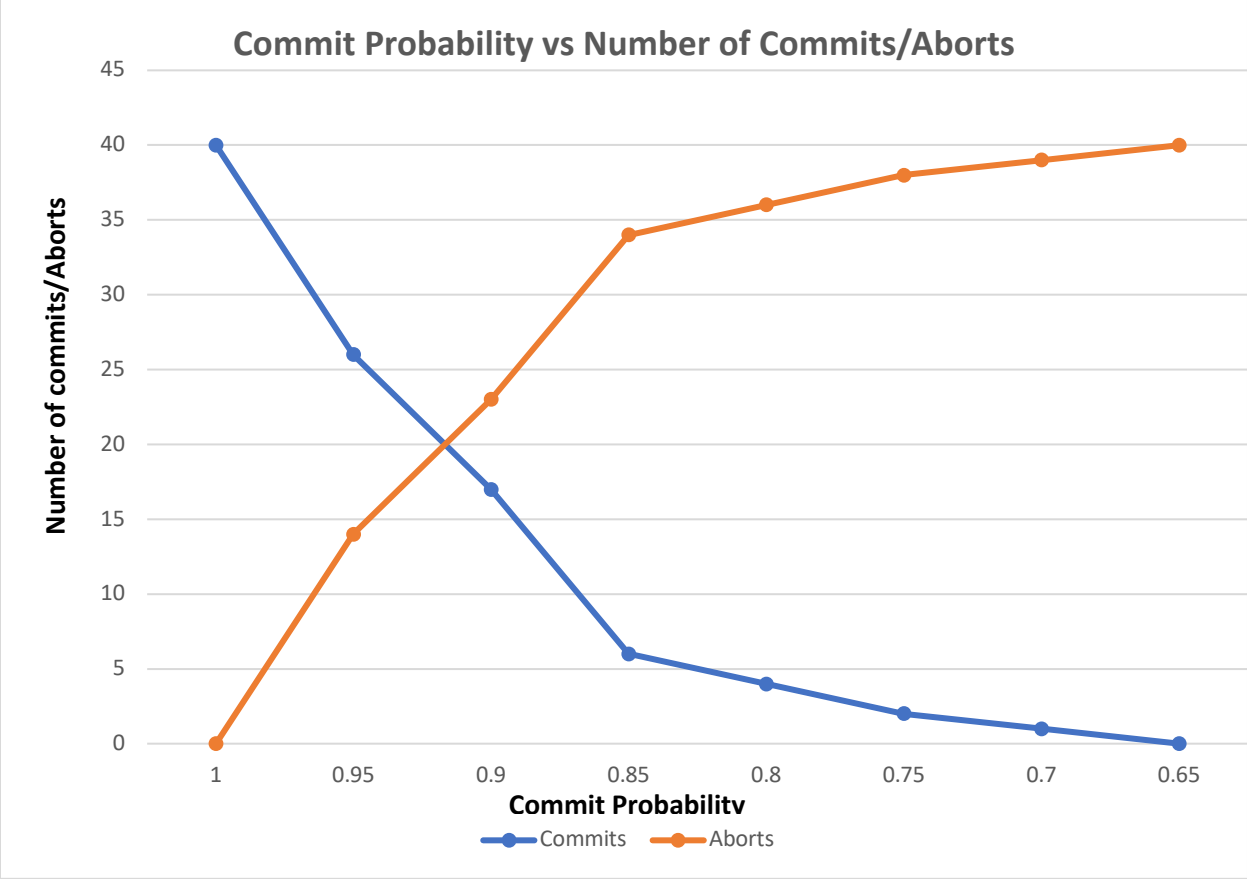
## Commit/Abort Probability

As we vary the commit and abort probability, we can see big increase in the number of total aborts. As multiple participants make independent decisions, we can see that the final decision which is a combination of all the individual decisions leads to a lot of aborted transactions.

Command used –

./target/debug/two_phase_commit -c 4 -p 10 -r 10 -s <1-0.65> -m run

| | 1 | 0.95 | 0.9 | 0.85 | 0.8 | 0.75 | 0.7 | 0.65 |
|---|---|---|---|---|---|---|---|---|
| **Commits** | 40 | 26 | 17 | 6 | 4 | 2 | 1 | 0 |
| **Aborts** | 0 | 14 | 23 | 34 | 36 | 38 | 39 | 40 |



## Participant Message Send Probability

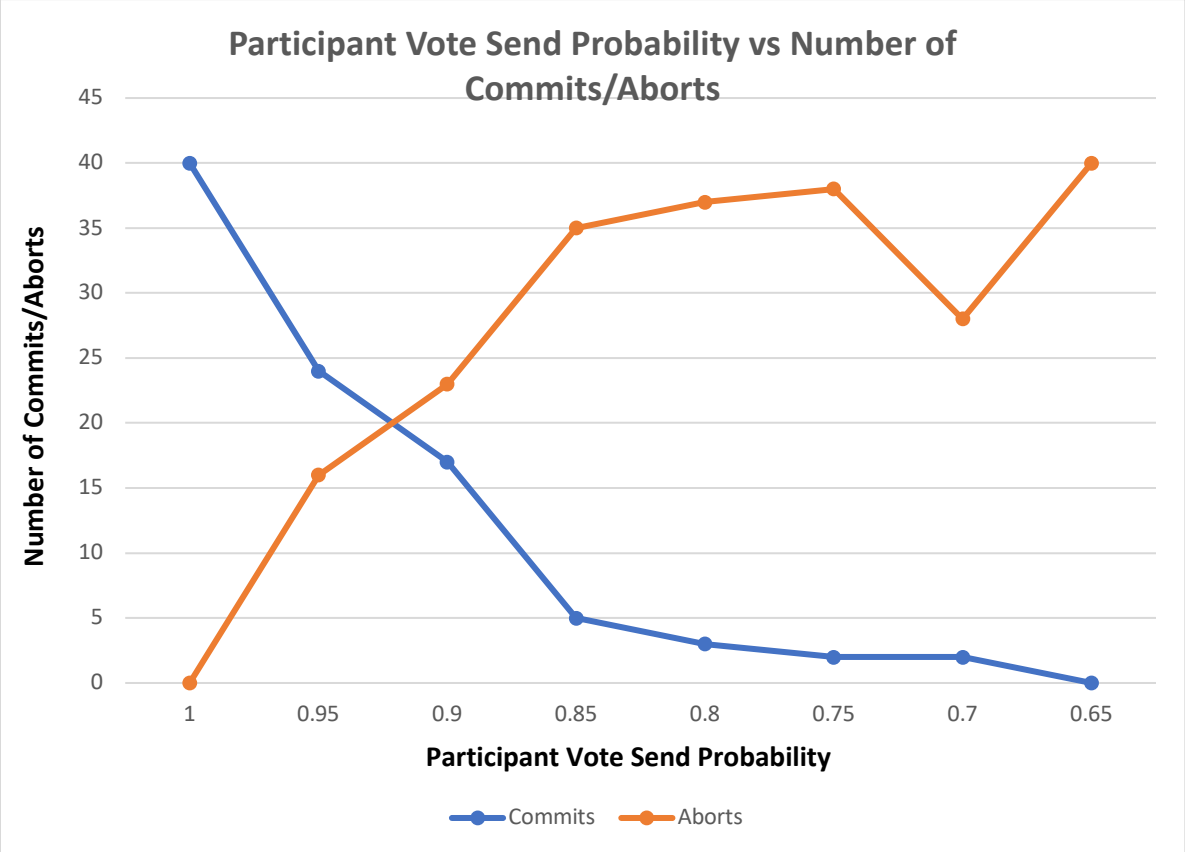
Reliability of the underlying infrastructure to transport messages is an important factor in determining the performance of 2-phase commit.

When we use the -S parameter to vary the probability of the message getting sent from the participant to the coordination in the 2-phase commit, we can see a quick dip in the number of committed transactions. This is one of the big bottlenecks with 2-phase commit.

Command used –

./target/debug/two_phase_commit -s 1 -c 4 -p 10 -r 10 -S <1-0.65> -m run

| | 1 | 0.95 | 0.9 | 0.85 | 0.8 | 0.75 | 0.7 | 0.65 |
|---|---|---|---|---|---|---|---|---|
| **Commits** | 40 | 24 | 17 | 5 | 3 | 2 | 2 | 0 |
| **Aborts** | 0 | 16 | 23 | 35 | 37 | 38 | 28 | 40 |



**Technical difficulty faced and insight gained**

In this project, number of things were new to me at a high level -

- Distributed transactions model
- Rust programming language.

The lectures were helpful is providing a base to the distributed transactions model. I later made use of the Gray and Lamport paper to better understand the transaction model. This gave me a fair understanding of the distributed transactions and the issues faced in distributed systems.

Rust was a brand-new language for me. However, my familiarity with C helped quite a bit. I did some initial reading to understand a few concepts of the language and got started.

The technical difficulty faced was firstly to establish communication between the clients, coordinator and the participants. This took a while as I was new to rust channels and the IPC channels had a very less documentation. I spent quite a while in this part of the project.

Once I was able to get communication working, I was able to move forward faster as I was able to understand the 2-phase commit protocol.

After clearing out a few bugs in the code, I was able to get the checker passing for different inputs.

My main takeaways from this exercise have been:

- Getting a good understanding of the foundations of distributed transactions and systems.
- Converting protocol in theory to code and get it working has been a huge learning.
- Knowledge of programming/debugging in Rust.