Final Project – Operating Systems
Bao Nguyen
Cory Pitt
CS 3281
Prof. Dubey

An Implementation of an ATM Network Using Raft Consensus

Overview

The problem of transaction commit has been explored by many different consensus protocols. Consensus is a problem intrinsic to fault-tolerant distributed systems. The issue involves multiple nodes, or servers in this case, agreeing on values to store in the distributed database when multiple servers might fail. Two-Phase commit (2PC) is often thought of as the traditional commit protocol. As one might infer, 2PC consists of two phases: the first is a commit-request phase in which one node, the transaction manager, coordinates all of the transaction resources to commit or abort. Once the decision has been coordinated, the transaction manager finalizes the operation by alerting all transaction resources of the final decision. However, 2PC and even three-phase commit (3PC) give rise to several issues due to the blocking property that the algorithms allow. For instance, in 2PC the system must wait for all resources (other nodes) to finish one iteration. Thus, if one node is not working properly the execution can be blocked. Additionally, the coordinator is a single point of failure. 3PC adds on another phase to 2PC in order to circumvent some of these issues. 3PC adds on a prepare commit phase, in which it asks all nodes if they are able to commit a current transaction. If any nodes reply no, or fail to respond in a given time frame then there will be a timeout message and an abort message replied. This can be altered so that the commit decision is based on a majority of nodes rather than every single, although then there are additional issues that arise when there are network partitions. For these reasons, our group decided against using 2PC as the consensus mechanism for our distributed ATM network.

In choosing between a different consensus mechanism it became apparent that our two strongest choices included Paxos Consensus or Raft. Paxos was developed by Leslie Lamport in the 90s and has been implemented across many modern systems – one of the most notable being Google's Chubby Lock service. However, from the beginning of its history Paxos's complexity has been widely noted as one of its main drawbacks. In fact, the Raft Consensus protocol was designed for the specific reason of creating an alternative to Paxos that was much easier to understand. For this reason, we chose Raft for our consensus mechanism.

Below is an overview of how the Raft consensus mechanism works within our system:

1. (Assuming that five servers are already running) Elect a server to be the leader server
2. The leader server appends the transaction logs to followers, and helps followers catch up with the leader's current log
3. If a log has been replicated to a majority of servers, it will be marked as committed. At this time, a change to our replicated state machine can be applied and the corresponding server will respond to the client.

Below we will go into further discussion of our implementation details.

In order to use the program, the user first needs to download the appropriate files which can be located here:

https://github.com/cs3281/final-project-Bachsanity.git

Once the files have been downloaded. The user must install the appropriate library that is needed by the program. This can be done by:

pip install –r requirements.txt

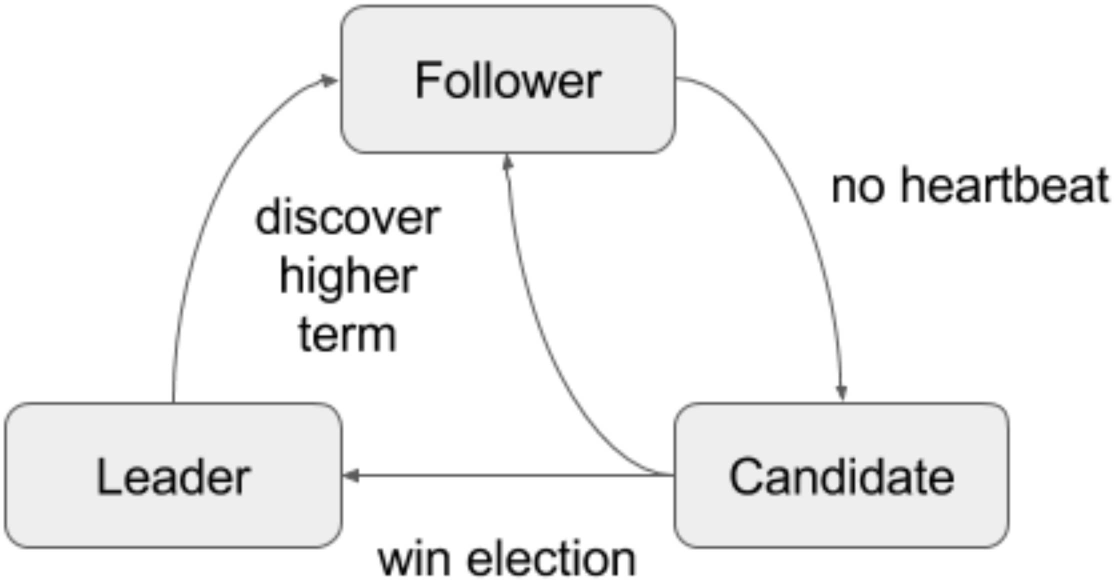
This is simply the installation of python's msg pack, which is necessary for our servers to be able to communicate (i.e. leader selection, transaction commit, etc)

The user should then enter the bank_teller directory. From the bank_teller directory the user first needs to begin running the appropriate servers (5000, 5001, 5002, 5003, 5004). On a mac this can be done by calling *sh run_server.sh <server port>*, from a linux machine this can be done with *./run_server <server port>*. If python 3 is not found, then change the run_sevrer.sh file to use python instead.

Once the program is listening from all five server ports, the user should run bank client by connecting it to a specific server. This is done by *python client.py <server port>*. Once the client is running from a specific port the user should be able to run the functionality of our ATM interface. At any time the user may shut down the client and any of the five servers. As long as there are a majority of the servers running (3/5) the program should continue functioning properly. Under these cases the system will be maintained until the majority of the servers are shut down.

Instruction on a mac although all functionality is provided for on a linux machine as well.
To test the program run sh.run_server.sh 500x, whereas x is one of the numbers {0,1,2,3,4}. This allows the user to access server ports 5000 through 5004. Each of these runs should be done from a different terminal window to demonstrate the five servers' requests in real time. Once all five servers are running then run python3 client.py 500x, whereas x represents the previously defined definition. This allows you to witness all client requests from its respective port's terminal window. Once this has occurred, you may shut down any of the terminals by closing its window. As long as 3 of the five nodes are running then the system will continue working correctly, even if the leader node shuts down. Once the leader is shut down, in a similar fashion to the start-up of the nodes, an election is run. The election process sends a vote request to each node. If a vote request is not responded to then this will time out. This happens in both the election process and the message consensus process. The time out process takes 5 minutes. However, if the server received the majority of the votes than it will become the leader and handle all client requests. This allows our raft implementation to show signs of

improvement over a 2pc implementation. While a 2pc implementation of such a network would fail if nodes became unresponsive and could not reply to the prepare message, a raft implementation will continue working. One might notice how this creates a central source for the system to rely on. However, the timeout will allow the new leader to be chosen and gives it it's non-blocking property. Below is a state diagram of our system.



The system doesn't not crash unexpectedly, however, occasionally it will throw a nonetype is not iterable error. This doesn't affect the functionality of the program however, so our group deemed the message has being ok.

Additionally

Implementation Details

Our project can be broken down by the three bank_teller files that enable our program to run and the Raft consensus mechanism.

1. Client.py
2. Run_server.py
3. Run_server.sh
4. Raft

Client.py

The Client file essentially acts as the interface for the ATM and handles all requests made by the user. Once there are the five servers running, the client is enabled to begin running the client from any of the servers.

This is done using UDP protocol with python built-in socket library. The socket will connect to a user defined server port. Once the socket has been connected then the user's transactions are able to be logged on the system. At any time, the user may terminate the client program by using the keyboard interrupt, although this will not affect the integrity of the database. At any time, the user may restart the client from the same server, or a different one as long as it is valid.

Run_server.sh

This will handle the start-up of each server. Due to the necessity of only having five different servers running we simply hardcoded the five different scenarios in which each of the servers on ports 5000, 5001, 5002, 5003, 5004 are handled individually. Additionally, this file will provide the necessary Raft package to the user's python path. To handle each servers start-up, each call to start a server will provide an additional call to run_server.py, which than can will perform the necessary actions to start the server from the specified port and connect it to our network of servers.

Run_server.py

This file begins running each server it is called upon and adds it to the network. The program defines the run_server method, which takes in a specified server call. The method first checks that the appropriate number of arguments was provided for the server (an appropriate server). The method will then make the specified server the local host and assign it a state of a follower, which is necessary to be used with the Raft consensus mechanism. Additionally, this file will use the asyncio library in order to get the event loop and synchronize our servers. The initial log the server is given is empty with dummy values set for its fields. Once the server is connected properly, the raft consensus mechanism takes over to ensure that the logs are updated appropriately and that there will be a single leader chosen that can handle the majority of the communication between nodes. If the leader fails then the mechanism also provides for a new leader election process. At any time, the user may terminate a server using keyboard interrupt.

Raft

In order to implement our Raft Consensus mechanism we used examples provided by a Github user named Reed and zhebrak here as a basis:

https://github.com/streed/simpleRaft

https://github.com/zhebrak/raftos

Our Raft implementation consists of three groups of classes: messages, servers, and states. Raft will use two main Remote Procedure Calls (RPC): Request Votes and Append Entries.

Messages: This group consists of five different types of messages that enable our system to communicate between nodes

1. Append_entries: This will simply create a constructor for a base message taking in self, sender, receiver, term, and data. This message is issued by the leader to replicate new log entries. An empty append_entries also acts as a heartbeat to maintain the authority of the leader
2. BaseMessage: This will define a base message with 4 properities being sender, receiver, data, and term
3. Request_Vote: This will either request a vote message (such as in the election of a new leader), or request a vote response (to gather the votes from the nodes)
4. Response_Message: An alternative constructor for a message that uses base message as a base and then gives it the property of a response when necessary. This will be used by followers to respond to requests from the leader
5. Serializer: This class defines our the serialize message which assigns data as having a message type, sender, receiver, data, and term and returns a msgpack given the data. The class also defines deserialize which unpacks a message and handles its type accordingly. Additionally, it can serialize or deserialize a client, which uses msg packs in a similar fashion although the data is the command and client port. The serializer is needed to convert data into a bytestream to send it over the UDP protocol. Separate methods are needed for client request and normal request because the format for a client request is different.

Server:

This class handles a server and the logs that must be distributed between each of the nodes. The class has the functionality of starting a server, broadcasting a message, requesting a message response, posting a message, and putting a message on the server. The server (node) can be in one the three states: Follower, Candidate, or Leader. There is also a UDP Protocol class define in order to sync messages to be sent between the servers. This class must check if a connection between servers has been made and whether a message has been received. Lastly, another USP server class is defined which creates a thread in order to wait for a message from the user client.

States

This group of classes define the individual states that a server may be in. They include base_state, candidate, follower, leader, timer, and voter.

1. Base state: the abstract class for all states to be further defined as needed

2. Candidate: A Raft candidate for election as leader. This is a transition state between follower and leader, in which a node sends out vote requests and if it successfully receives a majority of votes then will be deemed the leader and modified accordingly. This class also define the beginning of the election process in which it increases the term, votes for itself, and then sends out other vote requests. If the node received an append entry from a leader or not enough votes than it resigns from the election, turning back to a follower.
3. Follower: The state of most nodes in the network. A follower will turn into a candidate when it timeouts without receiving a "heartbeat" from the leader. Otherwise it will simply listen to entries broadcasted by the leader.
4. Leader: The leader is responsible for ensuring the integrity of all transactions. Our implementation does not currently provide for a step-down feature, so the leader will remain the leader until its shut down and a new election must take place. The leader will receive client requests from other servers or itself and process them accordingly. It checks to make sure a majority of servers are running to process the transaction, checks the validity of a transaction, and then will broadcast the transaction to the other nodes. The leader also handles a majority of user errors such as debiting an account with insufficient funds. Additionally, the leader sends out heartbeats at a specified time interval to check for the liveliness of the system.
5. Voter: This is the base class definition for follower and candidate states. The class defines the needed functionality for a vote to take place such as what a server should do when it receives a vote request and how it can send a vote response message.
6. Timer: This defines the necessary functions for the synching of our network. To do so the timer class schedules periodic call backs and provides for the starting, stopping and resetting our nodes.

Main Challenges

Some of the main challenges we ran into in designing our project included running the servers as five separate processes than could communicate rather than threads, dealing with error handling, and shutting down individual servers at a time.

The first issue was resolved by changing our run_server.sh bash script to only start one server at a time. The script also takes command line arguments for the neighbor ports so that the server can know who are its neighbors~~.~~ We were then able to connect the servers for communication using sockets and the Raft consensus protocol for message design and handling when a server went off line. This actually solved our third issue as well, for once this and be done we could simply run each server in its our terminal window and use keyboard interrupt to shut the server down.

This then allowed to us to test our distributed system by inputting invalid inputs as well as shutting down individual servers at a time. We did this using an organized method, where we began running five servers and then would test user input, and then test functionality when servers were taken offline. We first performed these tests shutting down two of the five follower servers, and then restarting them to check that they were updated to the current state of the logs appropriately. We then tested shutting down the leader server and a follower server and were able to observe the election process successfully take place. We then restarted those servers which we again updated appropriately.

Other challenges we ran into include waiting for a client request without interrupting the “flow” between nodes, picking a good timeout time so that split vote does not happen too often, and how to update a restarted server’s log after it has been terminated.

We resolved the first problem by creating a daemon thread to wait for a client request. This is because if we put the client request on the same queue with all the other messages, the delay can be noticeable since there are a lot of communication (sending and receiving heartbeats) going on between our servers.

The current timeout was chosen based on suggestions by previous implementations and by trial and error. It results in fairly quick election of a leader, with a difference in term number of the new leader and the old leader around 5 to 10. However, sometimes split vote can still happen for fairly long and we need to wait for a noticeable time before a new leader is elected.

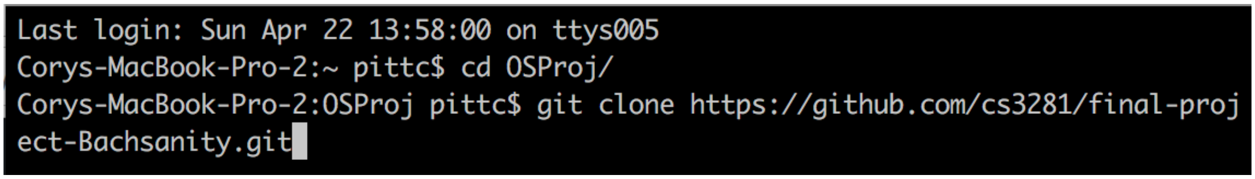
To make sure the restarted server’s log is updated by the leader, we have a nextIndex stored in the leader for each follower. This index indicates the next log index to be sent to the follower in a heartbeat. If the follower responds with a failure, it means that the follower is not updated and the leader needs to decrease the nextIndex for that follower. This will continue to happen until the follower finds an index that matches its own latest log index.

Division of Work

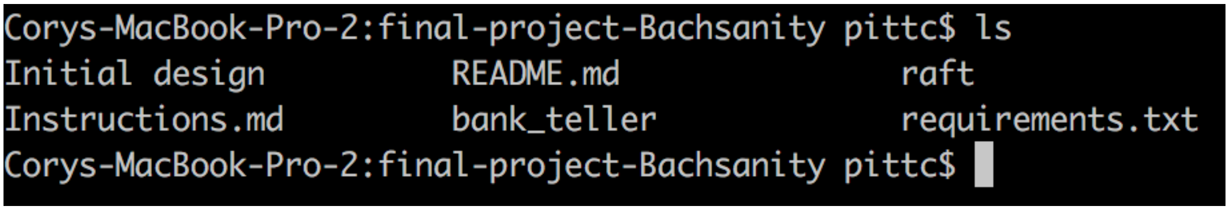
In order to complete our project in an efficient manner we needed to divide some of the tasks to build our system. Our initial design was created together after we each researched conventional consensus algorithms. Bao’s focus was the Raft implementation while Cory focused on testing the connections between servers and clients. Cory then wrote the final report and the analysis of the 2PC article for our project.

Results

The result of our project was a functioning network of five servers that could maintain a single client’s ATM balance, with operations to credit, debit, and query the balance. The system uses an implementation of the Raft consensus protocol. Below are images of our programs execution on a Mac operating system:

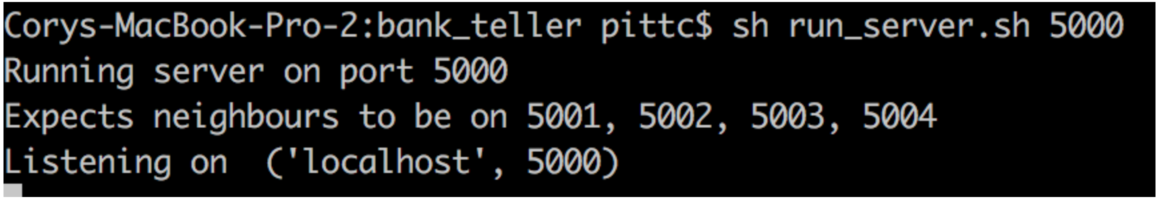


```
Last login: Sun Apr 22 13:58:00 on ttys005
Corys-MacBook-Pro-2:~ pittc$ cd OSProj/
Corys-MacBook-Pro-2:OSProj pittc$ git clone https://github.com/cs3281/final-proj
ect-Bachsanity.git
```



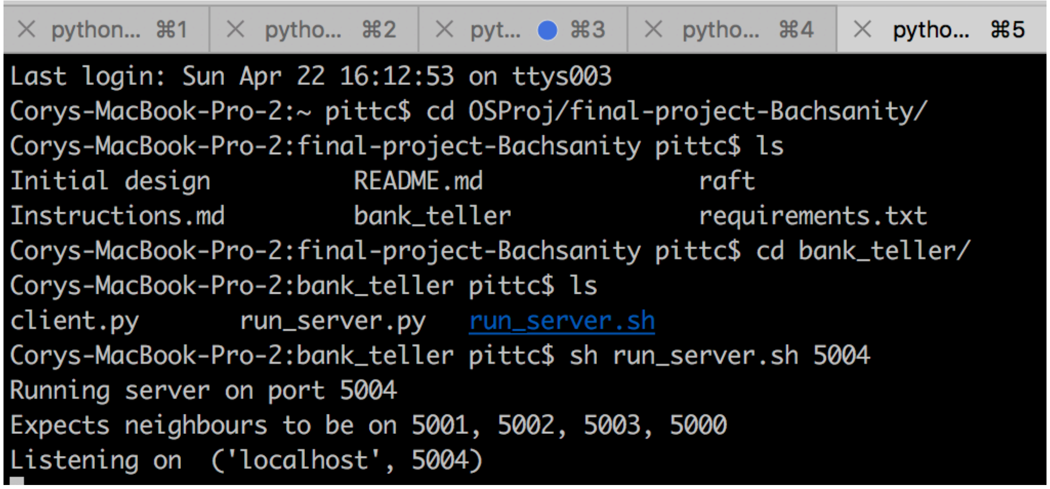
```
Corys-MacBook-Pro-2:final-project-Bachsanity pittc$ ls
Initial design          README.md               raft
Instructions.md         bank_teller             requirements.txt
Corys-MacBook-Pro-2:final-project-Bachsanity pittc$
```

Example of starting up a server



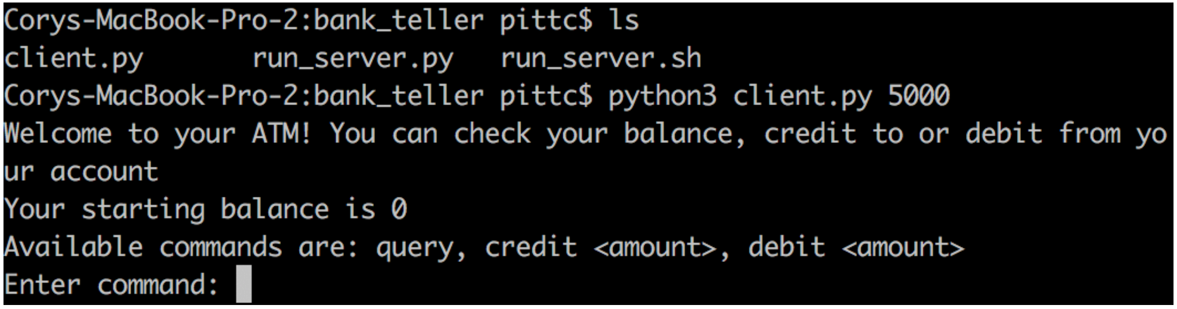
```
Corys-MacBook-Pro-2:bank_teller pittc$ sh run_server.sh 5000
Running server on port 5000
Expects neighbours to be on 5001, 5002, 5003, 5004
Listening on  ('localhost', 5000)
```

Example of starting up the five servers each with its own terminal window



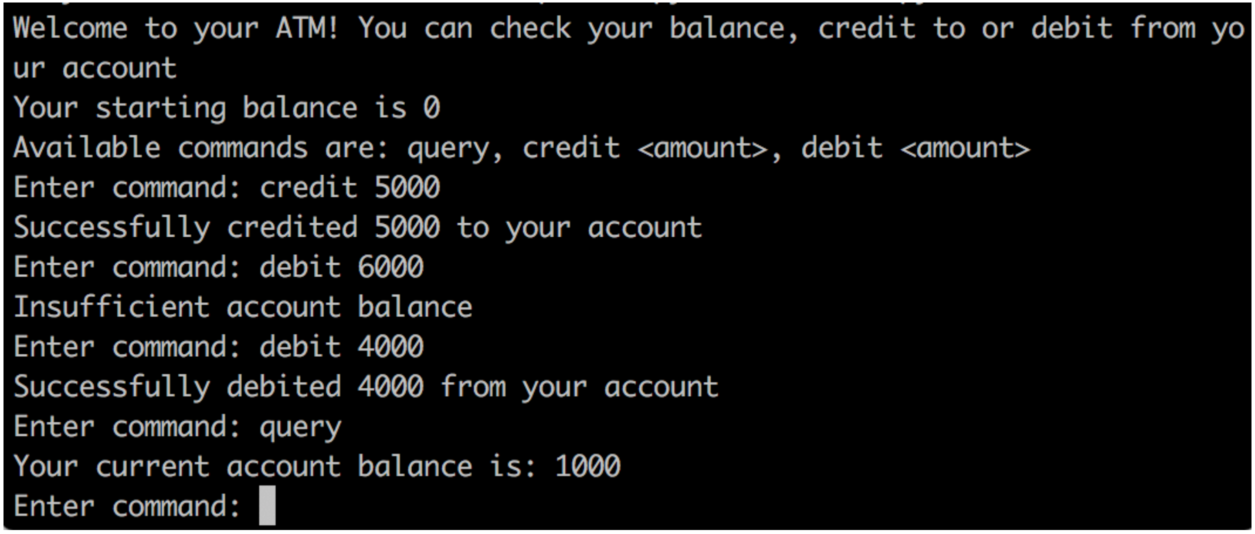
```
Last login: Sun Apr 22 16:12:53 on ttys003
Corys-MacBook-Pro-2:~ pittc$ cd OSProj/final-project-Bachsanity/
Corys-MacBook-Pro-2:final-project-Bachsanity pittc$ ls
Initial design          README.md               raft
Instructions.md         bank_teller             requirements.txt
Corys-MacBook-Pro-2:final-project-Bachsanity pittc$ cd bank_teller/
Corys-MacBook-Pro-2:bank_teller pittc$ ls
client.py       run_server.py   run_server.sh
Corys-MacBook-Pro-2:bank_teller pittc$ sh run_server.sh 5004
Running server on port 5004
Expects neighbours to be on 5001, 5002, 5003, 5000
Listening on  ('localhost', 5004)
```

Example of client interface



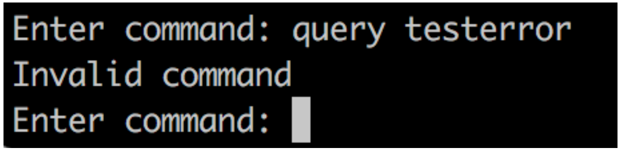
```
Corys-MacBook-Pro-2:bank_teller pittc$ ls
client.py       run_server.py   run_server.sh
Corys-MacBook-Pro-2:bank_teller pittc$ python3 client.py 5000
Welcome to your ATM! You can check your balance, credit to or debit from yo
ur account
Your starting balance is 0
Available commands are: query, credit <amount>, debit <amount>
Enter command:
```

Example of various inputs



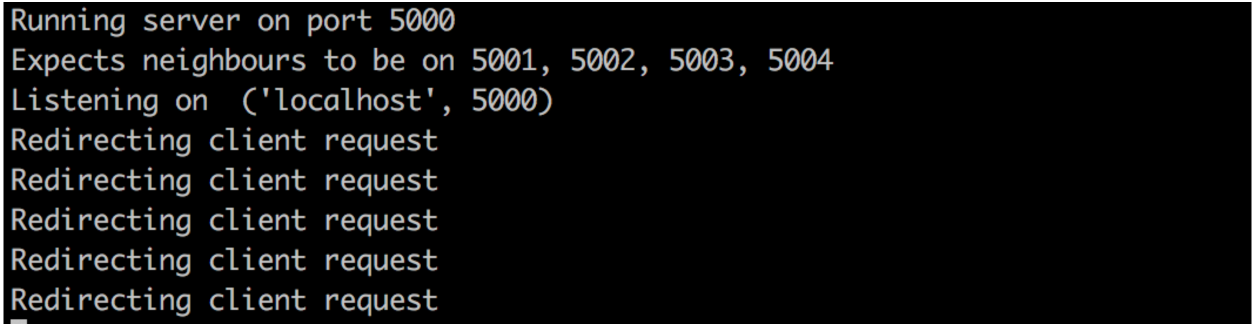
```
Welcome to your ATM! You can check your balance, credit to or debit from yo
ur account
Your starting balance is 0
Available commands are: query, credit <amount>, debit <amount>
Enter command: credit 5000
Successfully credited 5000 to your account
Enter command: debit 6000
Insufficient account balance
Enter command: debit 4000
Successfully debited 4000 from your account
Enter command: query
Your current account balance is: 1000
Enter command:
```

Example of an invalid input



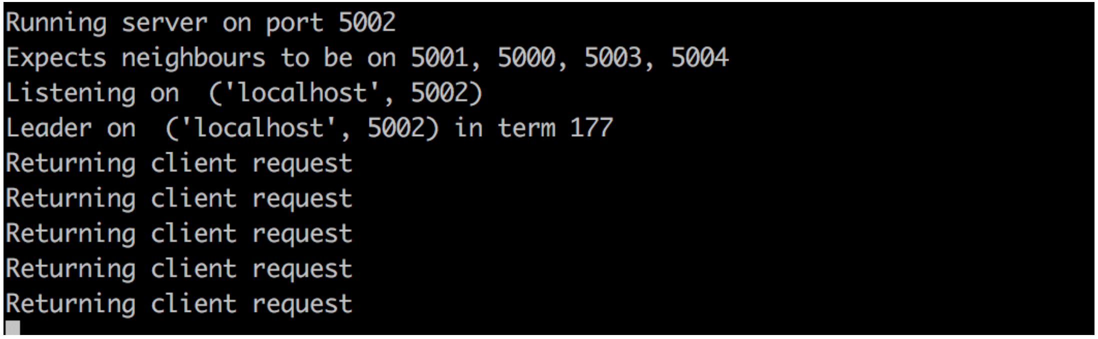
```
Enter command: query testerror
Invalid command
Enter command:
```

Example of the client server redirecting client requests to the leader



```
Running server on port 5000
Expects neighbours to be on 5001, 5002, 5003, 5004
Listening on  ('localhost', 5000)
Redirecting client request
Redirecting client request
Redirecting client request
Redirecting client request
Redirecting client request
```

Example of the leader handling client requests



```
Running server on port 5002
Expects neighbours to be on 5001, 5000, 5003, 5004
Listening on  ('localhost', 5002)
Leader on  ('localhost', 5002) in term 177
Returning client request
Returning client request
Returning client request
Returning client request
Returning client request
```

An Analysis of *Your Coffee Shop Doesn’t Use Two-Phase Commit*

Two-Phase Commit is thought of as the original protocol for transaction commit in distributed systems. An example of such as transaction commit is whether or not to commit an ATM transaction that occurs in one location to the entire system. This was a problem that was explored by our project, although we decided to use the Raft Commit Protocol. It’s important to consider certain aspects of the transaction, in order to shine more light on what it means to commit or abort a transaction to a database. For instance, if an ATM debit transaction is attempted, then the balance of that individual must be checked for sufficient balances. Once checked, if there are sufficient funds then that particular node (the current ATM) must notify all other ATMs in the system (other nodes) that the transaction took place. These are done through prepare messages that ask the network if they are able to undergo such a transaction (running properly, sufficient funds, etc), which is then responded with a Prepared message. Once enough prepared messages are received or not received the node will issue either a commit or abort message respectively, thus allowing the transaction to take place and the distributed database to be updated. In my opinion, the simplest way of thinking of transaction commit is planning to go to lunch with a group of friends, if you only wish to have the lunch if all of your friends are able to go. First you must call each friend individually and ask if they are able to go to lunch at the specific time (prepare message). If each friend says yes (prepared message), then you must call each friend back and confirm that everyone has said yes and the lunch is a go (commit message), else let them all know that the lunch has been canceled (abort message).

For our project, we were also asked to interpret the significance of a paper, *Your Coffee Shop Doesn’t Use Two-Phase Commit* by Gregor Hohpe, regarding 2PC and variations of commit protocols as they apply to real world applications – such as waiting in line at a coffee shop. The paper discusses the differences between concurrent and asynchronous tasks that take place during a transaction, optimizations and exception handling. The paper’s underlying theme is demonstrating how a Starbucks actually operates, in contrast with how it would operate if it were functioning under the pretense of 2PC. To do so, Hohpe discussed how when a customer places an order the cashier places it into a queue and then takes the next customers order. If the business was using 2PC then the cashier and customer would have to wait until the coffee is prepared to take the next customers order. Additionally, he highlights how the asynchronous process that is actually used increases efficiency, but also creates additional problems. One of these problems is that the drink orders don’t necessarily need to be fulfilled in the sequence they were placed. This allows for other efficiencies such as batch fulfillment, but there must be some sort of identifier that allows the order to be matched with the customer. Additionally, Hohpe then elaborates on how different exception handling techniques come into play in systems, by using the coffee shop example to break down the different options. These include Write-Off, Retry, Compensating action, and transaction coordinator. Each of the different options has its own benefits and drawbacks but can be thought of in terms of real world events. However, each strategy differs from 2PC commit except that which uses a transaction coordinator. Hohpe then dive into how in the coffee shop a 2PC system would operate extremely poorly, but in certain transactions with high stakes such a protocol can be necessary.

Such as in the purchase of a home using an escrow company that ensures parties meet their obligations before enabling the transaction to actually take place.

Finally, the paper discussed synchronous and asynchronous systems. To demonstrate his point Hohpe uses an example from Amazon's market design. There are certain situations which require that multiple nodes are in synch such as a conversation or the interaction between a customer and Amazon's site. However, once an order is actually placed the system can operate asynchronously with the order. As mentioned before, this allows for such optimizations as batch processing.

Overall, we believe the main point of this paper is to demonstrate how protocols such as transaction commit can be related to the actual world around us. Developed business processes will sometimes show us opportunity to optimize certain algorithms and protocols. Reading this paper made me consider a particular process that I have been undergoing lately. Currently, my tap water dispenser is broken in my fridge. This means that my current source of drinkable water is the sink, which does not produce cold water. Hence, recently I have been filling up cups of water and putting them in the fridge. However, I've discovered that the optimal number of cups in this "process" is two. Each time I drink a cup of water I simply put another one in the fridge. This allows me to avoid the time constraint of waiting for the water to cool. If I was only using one cup, then I would need to wait for an allotted amount of time to drink water, and if I want to drink water before doing anything else than I must wait for it to cool before completing another task. However, I could alternatively put many cups of water in the fridge. This again is not optimal, because it still takes the same amount of time to fill up each cup (just front loaded), and then I'm using unnecessary space in my fridge.

Overall, Hohpse's paper demonstrates to me that examples of process optimizations are everywhere in the world. It's up to individuals to identify such optimizations and translate them into efficient systems.

References:

https://renjieliu.gitbooks.io/consensus-algorithms-from-2pc-to-raft/content/index.html

https://github.com/streed/simpleRaft/tree/master/simpleRaft

https://medium.freecodecamp.org/in-search-of-an-understandable-consensus-algorithm-a-summary-4bc294c97e0d

https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14

https://github.com/zhebrak/raftos

http://www.enterpriseintegrationpatterns.com/docs/IEEE_Software_Design_2PC.pdf