Names: Aneesha Dasari and Veena Nalluri

VUnet IDs: dasaria and nallurv

Emails: [aneesha.dasari@vanderbilt.edu](mailto:aneesha.dasari@vanderbilt.edu) and [veena.nalluri@vanderbilt.edu](mailto:veena.nalluri@vanderbilt.edu)

Class: CS 3281 – Vanderbilt University

**Overview**

This project implements a bank ledger. Using the distributed consensus problem solution of the two phase commit protocol, this program simulates five different ATMS or instances that hold information about one bank account. An instance of the program runs by calling main and being given the port number for itself and the port number for the other instances. Each program, or ATM, supports three actions. The first is “credit” which will add money to the bank account. The second is “debit” which will remove money from the bank account if there is enough money. The third is “query” which will return the amount of money in the bank account. There is also a “help” command which will describe how to use the other three and an “exit” command which will exit the program. This project also handles the fault case of if an instance of the program terminates. When the instance, or ATM, is revived it will get the current balance of the bank account from other instances of the program.

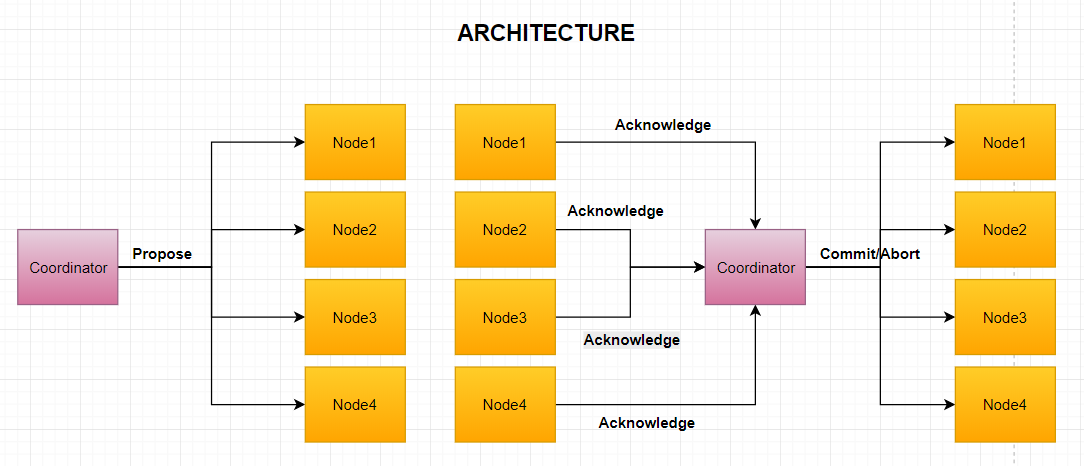
**Significance of “Your Coffee Shop Doesn’t Use Two-Phase Commit”**

This article called “Your Coffee Shop Doesn’t Use Two-Phase Commit” by Gregor Hohpe discusses looking at Starbucks’s models of fulfilling orders in order to design successful asynchronous solutions. Hohpe begins by looking at having a correlation identifier in messaging architecture. This is similar to having port numbers for different instances, or ATMS, for the project. Having the correct correlation identifier can make some end of the communication identifiable. He continues on to look at the topic of exception handling. This is very much like the handling of the fault case in this project. The closest example he gives to the implementation of one program instance terminating is the “retry” where the program just retries a failed attempt. Although the fault case in this program handles a terminated program being revived, the revived program does use the two-commit protocol to get the balance again.

Hohpe suggests that although a two-phase-commit protocol is simpler, it is more rigid in the flow of many asynchronous actions. Sometimes this can be good, and sometimes it is not. This very much depends on the stakes as well as the desired outcomes. Hohpe uses an escrow company as an example, looking at crediting and debiting money, very similarly to the way this project implements a bank ledger. However, the success of the two-phase-commit protocol again depends on every detail of a system and the desired outcome. He does talk more about the “conversations” that happen for transactions, and how those are synchronous. In the context of this project, we see synchronous communication between different instances of the program using sockets.

Overall, Hohpe proposes that a half-sync, half-async model be used. He looks at asynchrony as subjective, and that with specifically communication, there is some synchronous aspect to the system. This is seen in the current project with communication between ports and receiving and executing transactions. This article makes us think about the asynchronous model that we have implemented, and suggests that there may be a better implementation by understanding and recognizing how asynchronous a model really is.

**Architecture**

****

**Implementation Details**

We implemented this by having a main function and a function called receiving\_ports which implements the two phase commit protocol. Within the program we have a few important global variables: the balance of the bank account and a value called acknowledge that tracks the number of acknowledgements from other instances of the program.

The main function takes in the arguments that are given to the program, which are the first port number (the coordinator port) and the rest are the connecting ports. It then creates a socket and binds it to an arbitrary address. It then creates a thread to send and receive commands which enters a loop to do so. This loop takes in a command and figures out what command was given to it to act accordingly. This is implemented through an if statement. This instigates sending signals through the sockets to other ports. The signal depends on the specific command. The receiving\_ports function works with getting the acknowledgments and handling the signals that are received from the sockets. It received the type of transaction and uses the two phase commit protocol using sockets to send “yes” or “no” for when being prompted by a different coordinator and to count acknowledgments and send “commit” to other ports for when it is the coordinator. This is mostly for the credit and debit actions. The other action that also uses the two-phase commit protocol is for the fault case. This uses “disconnect” and “setbalance” signals that are specified to get the balance from other ports and set it as its own.

**Main Challenges**

There were a few challenges to this project. The first was to figure out the specifics of the desired solution to the problem. It took a lot of research and asking questions to figure out exactly what was needed from this program, to figure out what should happen in different cases, and to figure out how this should work and can be implemented properly. We struggled a bit in trying to outline the frame of the program itself. It was difficult to take our working understanding of the program and figure out what files and methods we needed. We outlined something at the beginning of our work, that we later changed to be completely different. Another difficulty was implementing the sockets and the communication between sockets. Although we had an example in class, using sockets in the program, especially using C, proved difficult. We finally figured this out by using online forums. Finally, the division of work was hard. We found working remotely and evenly was difficult, as this was a smaller scale project. As such, we had to divide certain things up, and work together in person on one computer for other parts of the project.

**Division of Work**

As stated, this was difficult to divide every aspect of the project evenly. Although both people worked on all parts of this project, Veena put in an effort on the working base of the project with much assistance with implementation, debugging, and cleaning up code from Aneesha. Aneesha worked more on the report and presentation preparation, with assistance with writing, proofreading, and preparation from Veena. Overall, division of total time and effort ended up pretty evenly split in all parts of the project.

**The Result**

The result is a working prototype. Opening up five different terminals and beginning five instances of the program, a user can issue commands from any of these instances to credit or debit to the bank account. The program could definitely be cleaned up even more to account for even more fault cases. However, the resulting program is a basic beginning for a bank ledger using the two phase commit protocol to solve the distributed consensus problem.

**Testcase1:** Open up all 5 terminals and provide following commands in each terminal. Here we are testing the basic functionality of providing credit, debit and query operations.

1. ./main -p 5600 -n 5601,5602,5603,5604

2. ./main -p 5601 -n 5600,5602,5603,5604

3. ./main -p 5602 -n 5601,5600,5603,5604

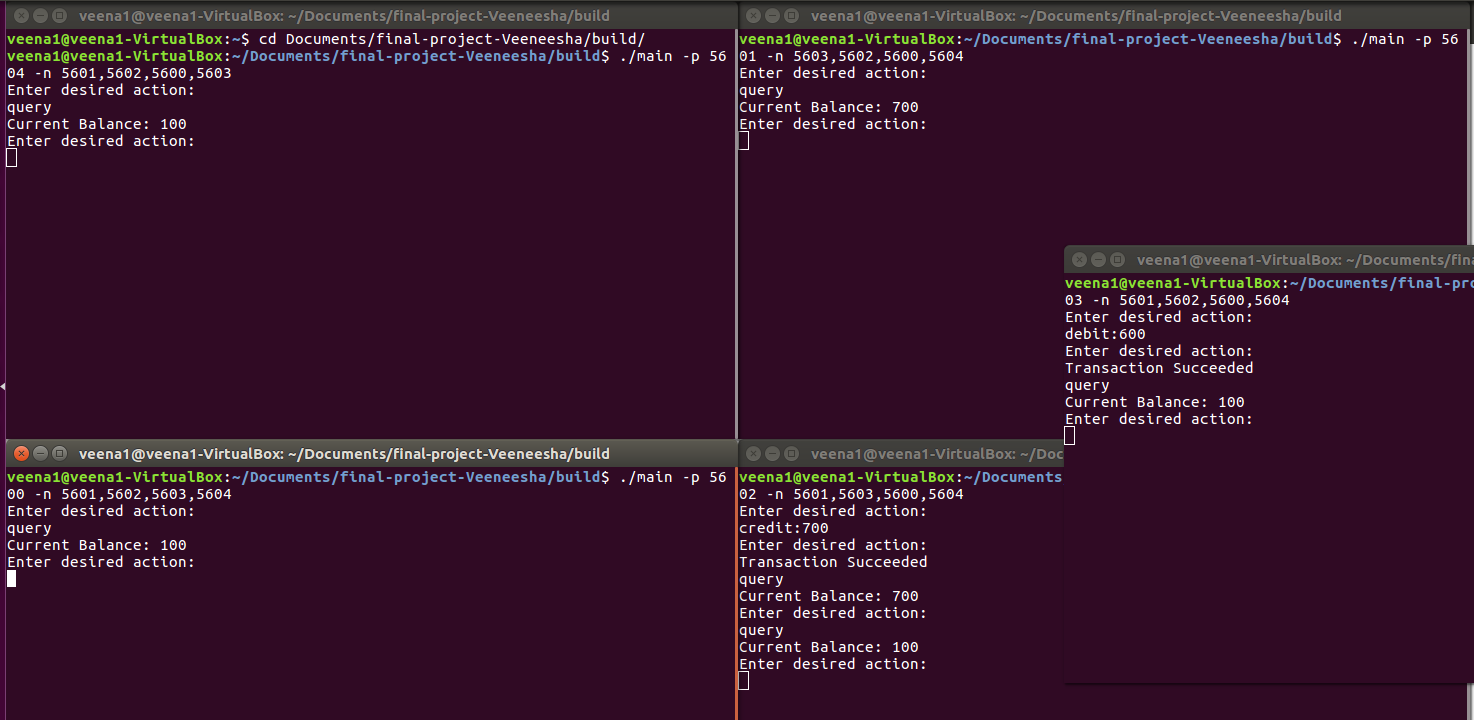
4. ./main -p 5603 -n 5601,5602,5600,5604

5. ./main -p 5604 -n 5601,5602,5603,5600

Testcase2: Implementing Fault case when a node terminates and reconnects it has to be in consensus with other ATMs, so we manually terminate node ./main -p 5601 -n 5600,5602,5603,5604 and run the node again to get the current balance.

\

Output for Basic Functionality:



Output for Fault case:

