Troy Toman
Dr. Dakai Zhu
CS5523 Operating Systems
April 26, 2010

PROJECT 4 FINAL REPORT

Overall Project Result

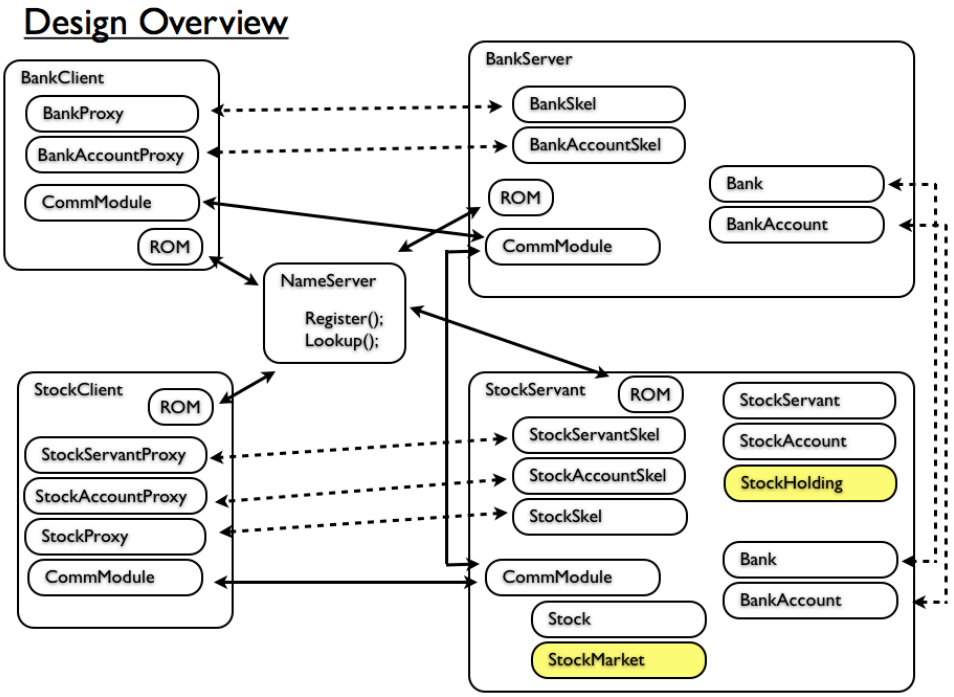
I was able to complete all of the requirements for the project including the Stock Client, Stock Server, Bank Client, Bank Server and Name Server. The project was coded in C++ and only used standard libraries including Pthreads and basic socket programming. Bank and Stock server transactions are completed in separate threads. Sensitive operations such as account creation, buying and selling stocks, and deposits and withdrawals are protected using Pthread mutex locks. A separate thread runs within the Stock Server to update stock prices on an ongoing basis. Most error handling has been implemented including handling non-existent stock and account lookups, attempt to withdraw or sell amounts not held in the accounts or recreation of existing account. The program also loads in 3 initial bank accounts and 2 initial stock accounts. There are five banks created and 25 Stocks supported in the final project.

The system does have some known limitations. While I attempted to test and handle erroneous user input in most cases, I cannot claim to have done this analysis for all possible user interactions. Also, to allow myself to remain focused on the distributed infrastructure, I simply used fixed size arrays for most lists in the system. The maximum number of accounts, for instance, is limited to 50. There are other hard limitations in this version of the system.

Final Design

My final design did not deviate significantly from the design presented in my initial design submission. I changed the name of some objects to help clarify the role each object plays. For instance, I changed StockServer in the Stock Client to StockServerProxy and StockAccount to StockAccount proxy. I also changed the StockServer object to Stock Servant.

The most significant change was to create a StockMarket object in the StockServant. This became necessary as way to keep consistent control of the price for Stocks. As such, I created 2 distinct kinds of stock objects. First, a Stock is an object that contains a symbol, company name and price. Then, I also have a StockHolding which points to a Stock object and then keeps track of a number of shares. Each StockAccount then has a list of StockHolding objects that are in the account.



The NameServer, RemoteObjectModule and Comm module worked well in the model. I was able to leverage these pieces in all applications. I also used a consistent RemObjectReference class that kept track of all remote objects in the system. Marshaling was done by concatenating remote references, class/object/method IDs and parameters into each remote invocation. The dispatcher is in each server that invokes a skeleton based on the class ID. The object ID and method ID are used by the skeleton to invoke the local method. Data is then re-marshaled into a string and passed back to the client.

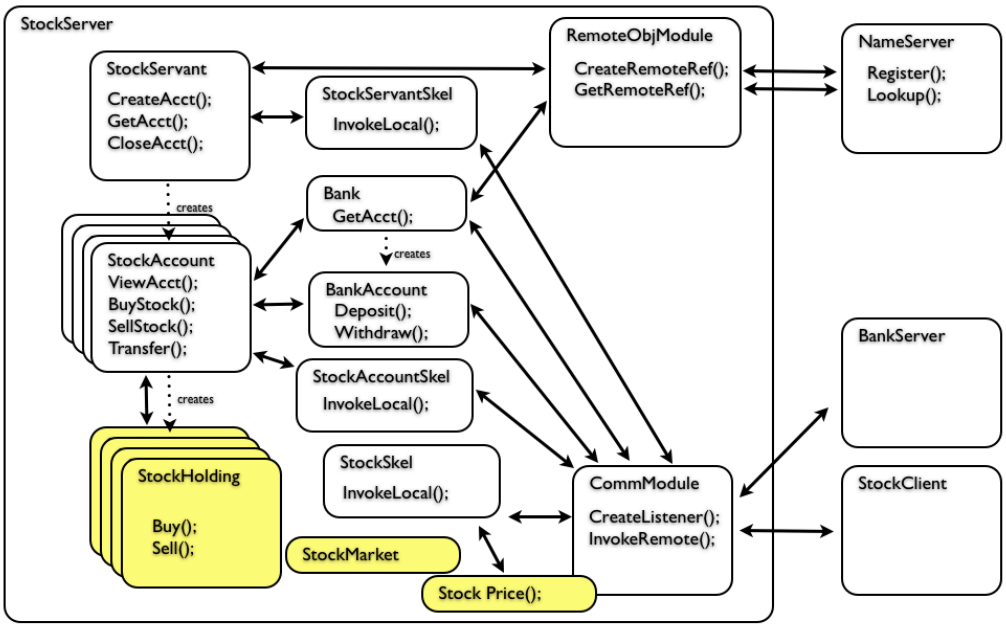
All user interaction is accomplished via a text-based user interface. I kept this very simple and focused most of my time on the "plumbing" of the system.

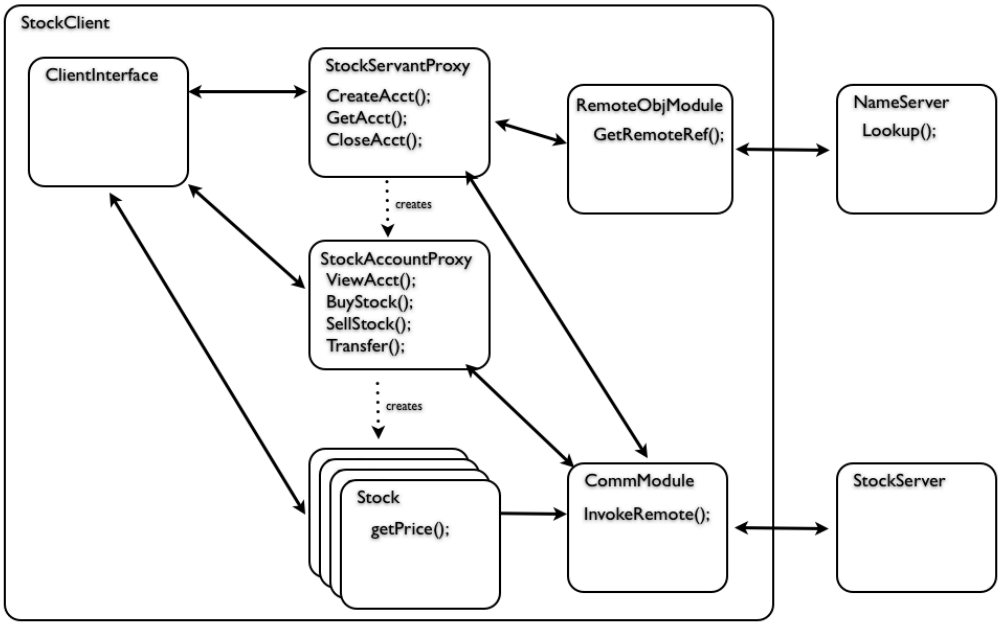
Project Testing and Results

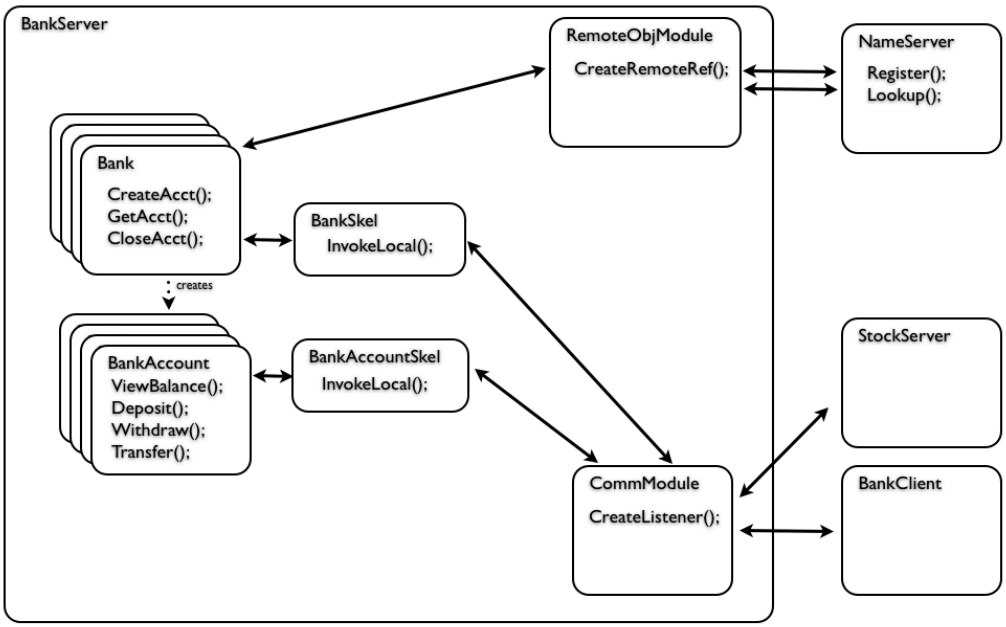
The project was developed and initially testing on a MacBook Pro and using the Xcode development environment. All initial builds were done with gcc 4.0. I began the project using gcc 4.2 but ran into a bug in the string library that has not yet been resolved on Mac OS X 10.6. The program works fine using the gcc 4.0 compiler.

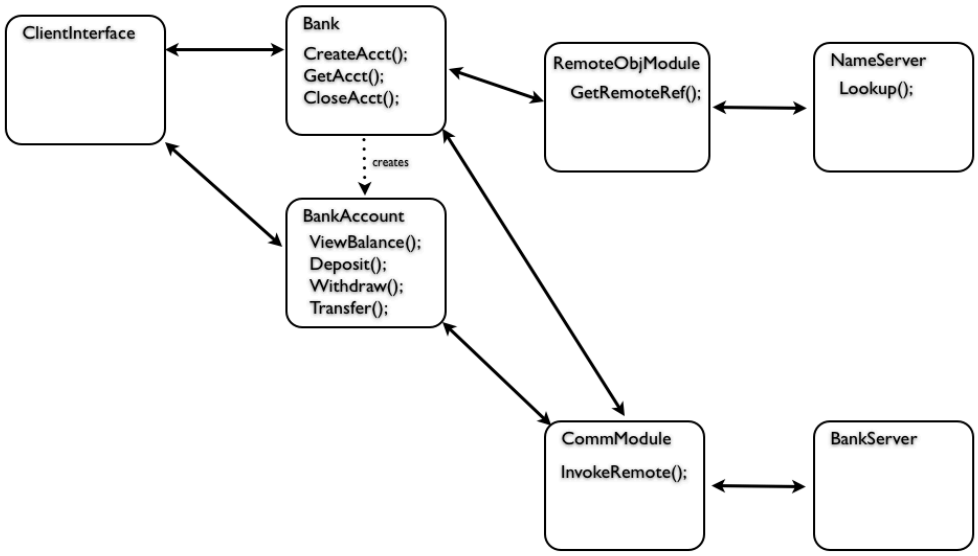
After completing initial testing on my laptop, I moved my server programs to a Rackspace cloud server. At this point I have successfully tested both the NameServer and the StockServer running on the cloud servers. I am hoping to demonstrate the system with each server running on a different system.

Testing has shown that the program works as designed. I have tested all of the basic interactions across all five programs and have validate the interactions between them. I have tried to test most situations where a user could enter incorrect data. I have found the error handling to work for these situations. However, I believe that I have probably not tested all possible errors in the system. So, there are likely scenarios that would cause the system to crash.


StockServer
StockServant
CreateAcct();
GetAcct();
CloseAcct();
StockServantSkel
InvokeLocal();
RemoteObjModule
CreateRemoteRef();
GetRemoteRef();
NameServer
Register();
Lookup();
creates
Bank
GetAcct();
creates
StockAccount
ViewAcct();
BuyStock();
SellStock();
Transfer();
BankAccount
Deposit();
Withdraw();
BankServer
creates
StockAccountSkel
InvokeLocal();
StockHolding
Buy();
Sell();
StockSkel
InvokeLocal();
CommModule
CreateListener();
InvokeRemote();
StockClient
StockMarket
Stock Price();


StockClient
ClientInterface
StockServantProxy
CreateAcct();
GetAcct();
CloseAcct();
RemoteObjModule
GetRemoteRef();
NameServer
Lookup();
creates
StockAccountProxy
ViewAcct();
BuyStock();
SellStock();
Transfer();
creates
Stock
getPrice();
CommModule
InvokeRemote();
StockServer


BankServer
RemoteObjModule
CreateRemoteRef();
NameServer
Register();
Lookup();
Bank
CreateAcct();
GetAcct();
CloseAcct();
BankSkel
InvokeLocal();
creates
BankAccount
ViewBalance();
Deposit();
Withdraw();
Transfer();
BankAccountSkel
InvokeLocal();
StockServer
CommModule
CreateListener();
BankClient


ClientInterface
Bank
CreateAcct();
GetAcct();
CloseAcct();
RemoteObjModule
GetRemoteRef();
NameServer
Lookup();
creates
BankAccount
ViewBalance();
Deposit();
Withdraw();
Transfer();
CommModule
InvokeRemote();
BankServer