CS3004 Network Computing Assessment

Dylan Leonard / 2156359

# Introduction

This report documents the requirements, design, implementation and testing of a client-server banking system for CS3004 coursework assignment. It will demonstrate my understanding of the main issues related to network computing, ability to critically evaluate requirements and problems when designing and implementing network computing applications.

# Requirements

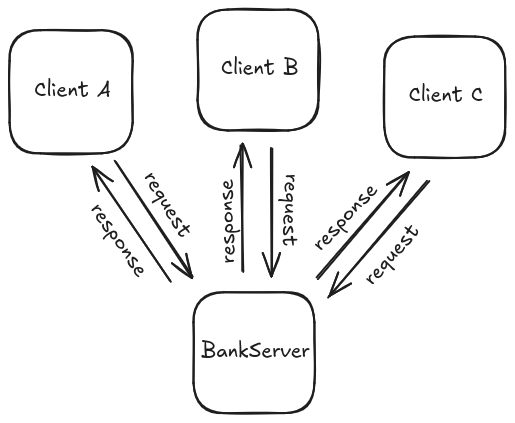
Create a client-server banking system that has allows for three client applications (A, B and C) to connect to a single, multi-threaded server that manages the client’s three operations: adding money, subtracting money and transferring money between accounts. The server must store three values (each starting at 1000 units) for each client with locking functionality to stop concurrency issues (lost updates, dirty reads, etc) and it must be multi-threaded to allow for multiple clients to connect at the same time. The server must produce logs to prove client requests have been processed.

# Design

This section will discuss how the design aspects including the architecture, protocol and client interaction.

## Client-Server Architecture

This banking system will follow the client-server architecture which is a model which contains servers which manage, store and provide data or services and the clients request the services or data from the server(s). As seen in the diagram below, the banking system will allow three clients to access a single server, sending commands (requests) and receiving confirmation responses. The server and clients will use the TCP/IP transmission protocol to send and receive reliable packets via Java’s standard java.net libraries (Socket and ServerSocket).



## Protocol Table

Below is the protocol table which describes the messages that are sent between the parts of the banking system. As the requirements state, multiple clients should be able to connect to the server at once so the table has been split into BankServer, the main server, and BankThread, an individual thread that manages a unique BankClient.

|  |  |  |
| --- | --- | --- |
| BankClient | BankThread | BankServer |
|  |  | [Run BankServer] |
| [Run BankClient] |  |  |
|  |  | WHILE NOT TERMINATED |
|  |  | [accept BankClient connection] |
| [successsful connection] | [BankThread started] | [start BankThread for connected BankClient] |
| WHILE NOT TERMINATED | WHILE NOT TERMNIATED |  |
| SEND command TO BankThread |  |  |
|  | RECIEVE command FROM BankClient |  |
|  | IF command IS VALID |  |
|  | [process command] |  |
|  | response = [command completion confirmation] |  |
|  | ELSE |  |
|  | response = [command is invalid] |  |
|  | END IF |  |
|  | SEND response TO BankClient |  |
| RECIEVE resonse FROM BankThread |  |  |
| END WHILE | END WHILE | END WHILE |
|  |  |  |
|  |  |  |

## Client Usage

As the client initiates communication, client will be able to send three valid commands:

add [value]

Adds value to the clients own account

subrtact [value]

Subtract value from the clients own account

transfer [value] [account]

Transfer value from clients own account to specified account in argument

Valid arguments:

[value] is a positive double

[account] is a case-insensitive valid account (A, B, or C)

# Implementation

This section will cover how all the key elements were implemented for the banking system.

## Server Start / Socket Opening

In the try-catch, the server socket tries to open port 4444 to TCP/IP connections using which allows clients to be able to connect to the server. If failed, sends an error and exits. This code snippet also shows the initialisation of SharedBankState which sets the private double array which stores the account values.

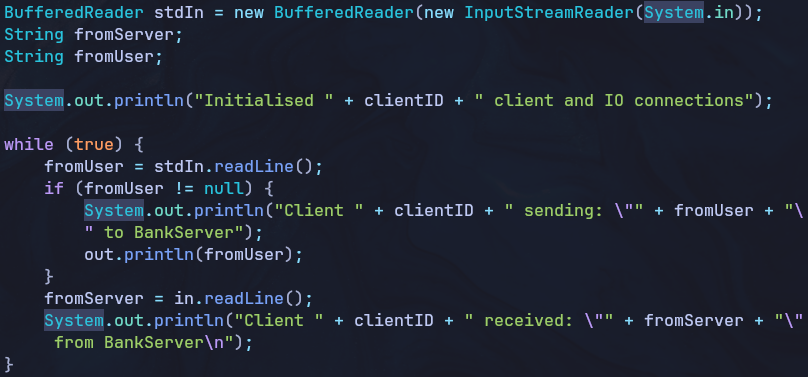
## Server Accepting Connections / New Thread Creation

After the server is successfully started, it will enter an infinite loop where it will check to see if a new client has requested to make a connection. Once a client makes a connection attempt, a new thread will be created to handle the client while assigning it’s account: A, B, or C (depending on the previous connection). This could be improved by the client specifiying which account it should be assigned to before connection.

## Client Init / Connection Request

The client tries to create a connection with the server socket with “new Socket()”. In this example, the hostname is “localhost” as the client and server run on the same system but a different hostname could be specified if the server is ran on a different system. If the connection is accepted, the input and output streams are connected to the server to allow for text to be sent between the client and server.

## Client Input Loop

Once connected with the server, reading clients input is initalised and the client enters an infinite loop of trying to read the client’s input. If the user has an input, it is sent to the server (along with logging to confirm transfer) and prints the servers response once received.

## Mutex Lock

A mutex (mutual exclusion lock) ensures that only one thread can access the shared state object which eliminated issues like dirty reads. When a thread tries to access the shared data (account values for the banking system), it will try to acquire the lock, if no other thread is accessing it, it will be given the lock and be allowed to access and modify the data. If a thread tries to access the data while it’s locked, it will wait until the thread releases the lock.

## 

## Validating Client Input

Regex was used to verify if the clients input was valid to allow for both capitalised and non-capitalised commands to be valid (along with account name). I used [0-9] instead of \d because Java requires two escape chars (\\d) so it was even more unreadable than it currently is but it simply checks if it’s a valid command as stated previously.

## Client Operation (add\_money, subtract\_money, transfer\_money)

Functions that execute the clients commands, interacts with the global private shared accounts array after acquiring the lock. “accounts” is the private global array of values for accounts. 

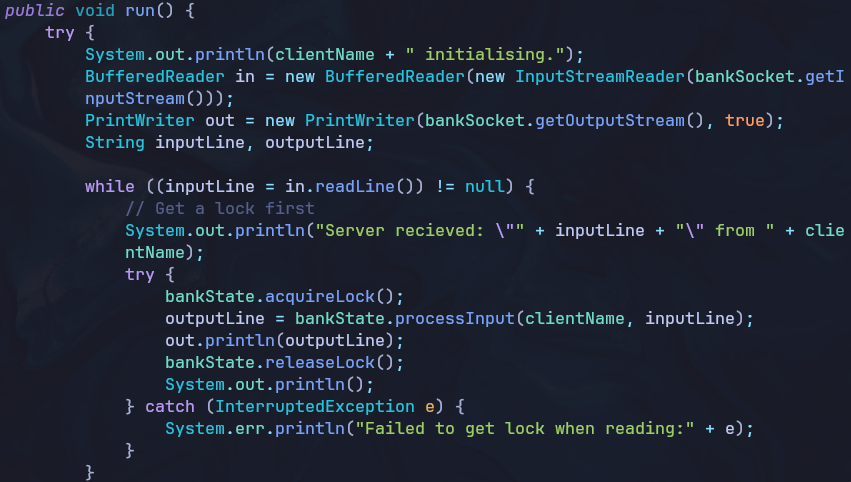
## Process Client Input

Function that processes the client input by validating the input, ensuring the thread was correctly initalised and if both pass, evalutates which command the client has sent and executes it. After execution the server responds to the clients request (command) by confirming it’s result. Also prints all accounts for server logs.

## 

## Bank Thread

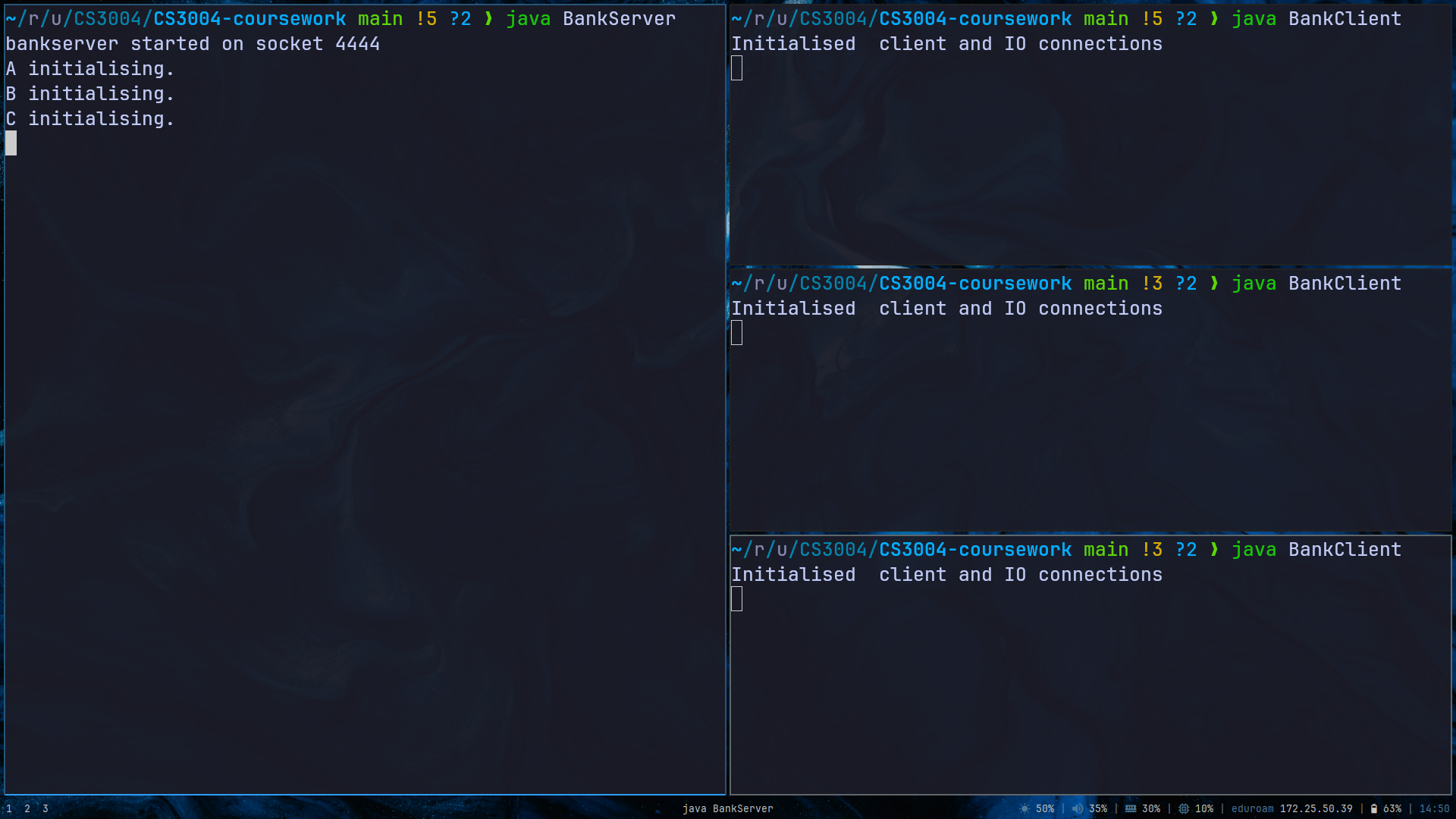
Started when the client tries to connect to the server, continues the connection process by connecting to the client’s I/O streams and when a request from the client is received, the thread tries to acquire a lock, process the request, and release the lock upon completion.



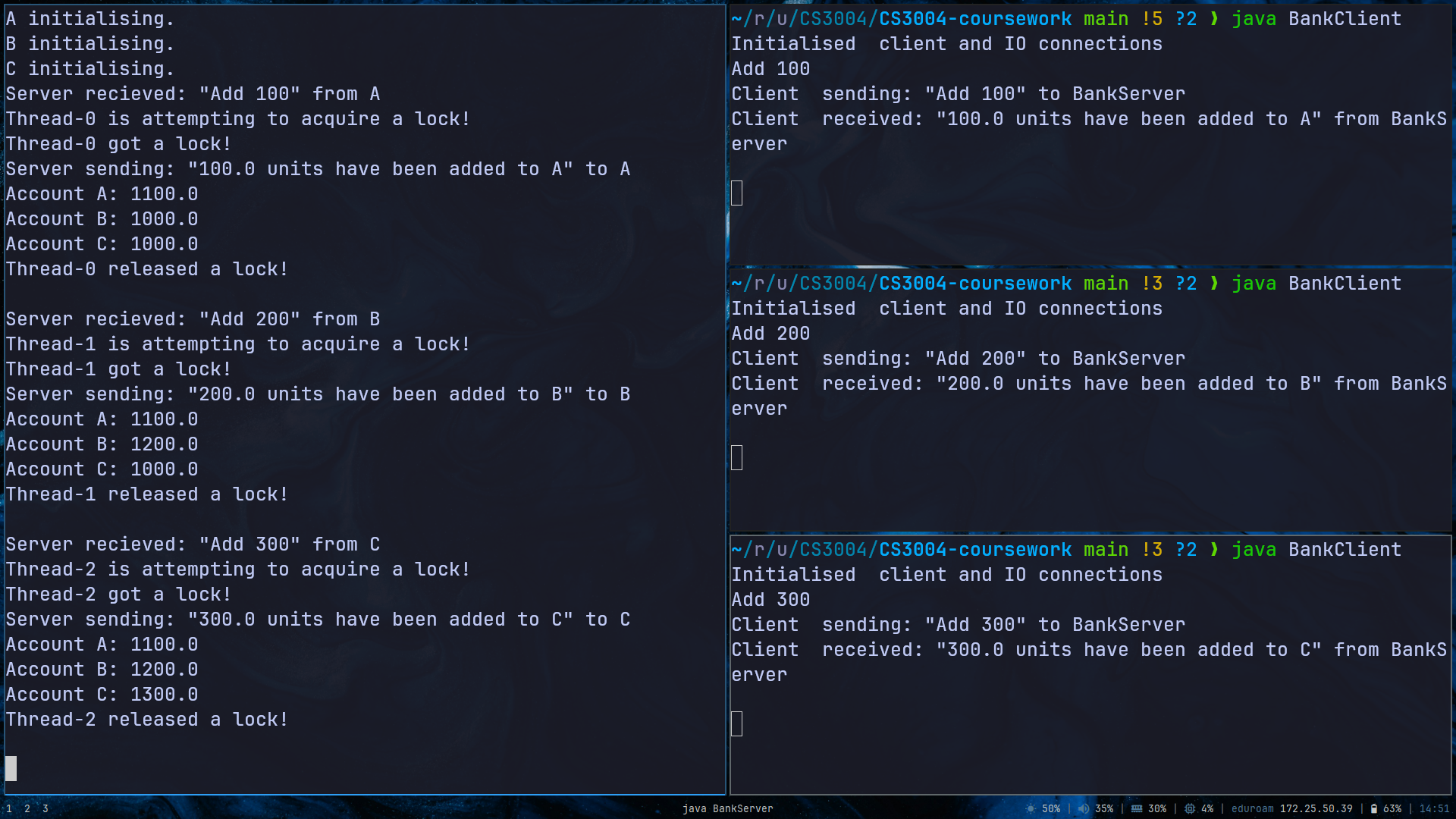
# Testing

This section will go over all the testing of the banking application. Each screenshot is the same format with the BankServer on the left, and Clients A, B and C respectively top-to-bottom on the right. (There are only one screenshot for each section as each shows the server (with logs), and three instances of the client performing the given command). As you will see, all the requirements have been completed.

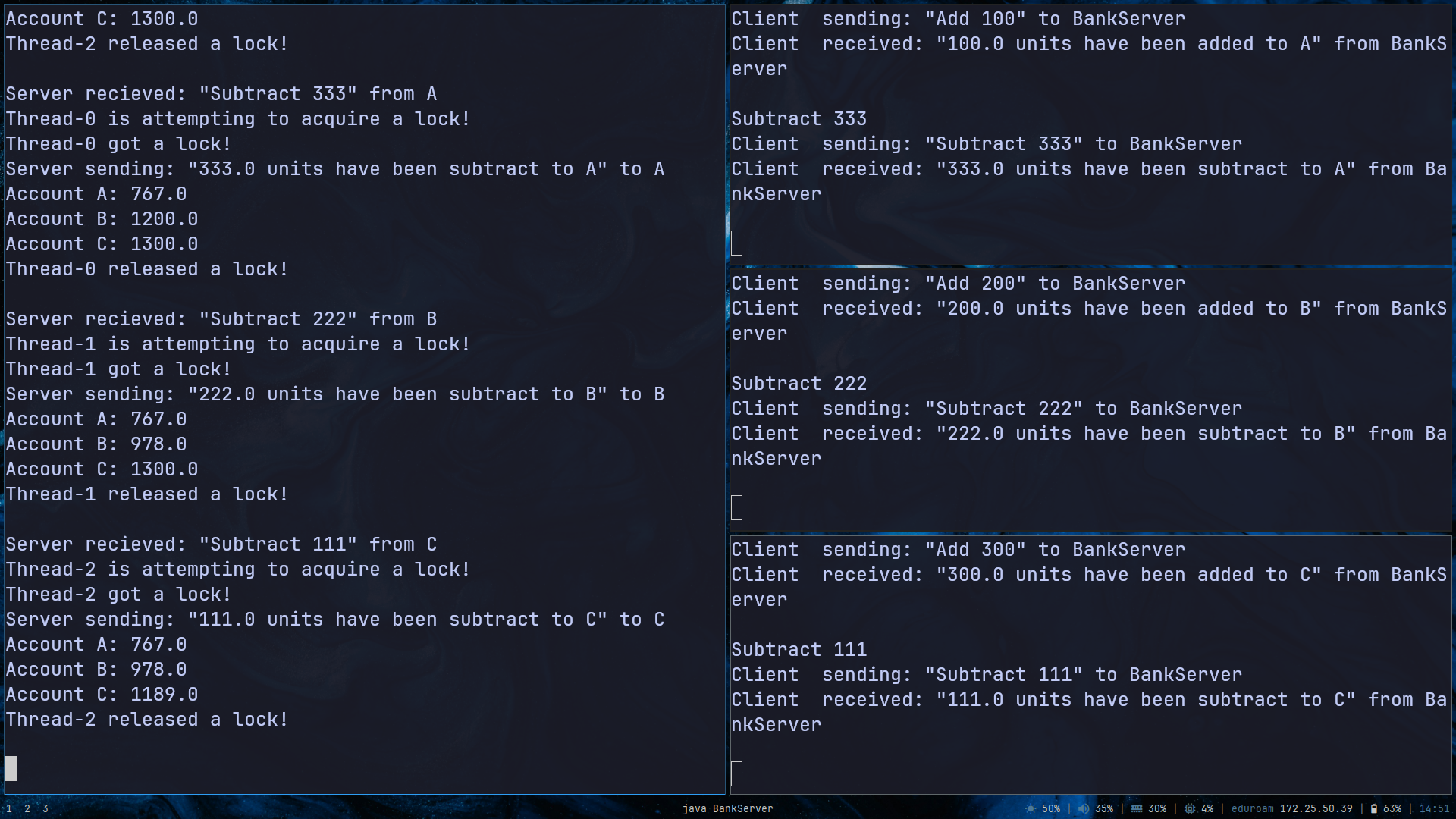
## Initialising



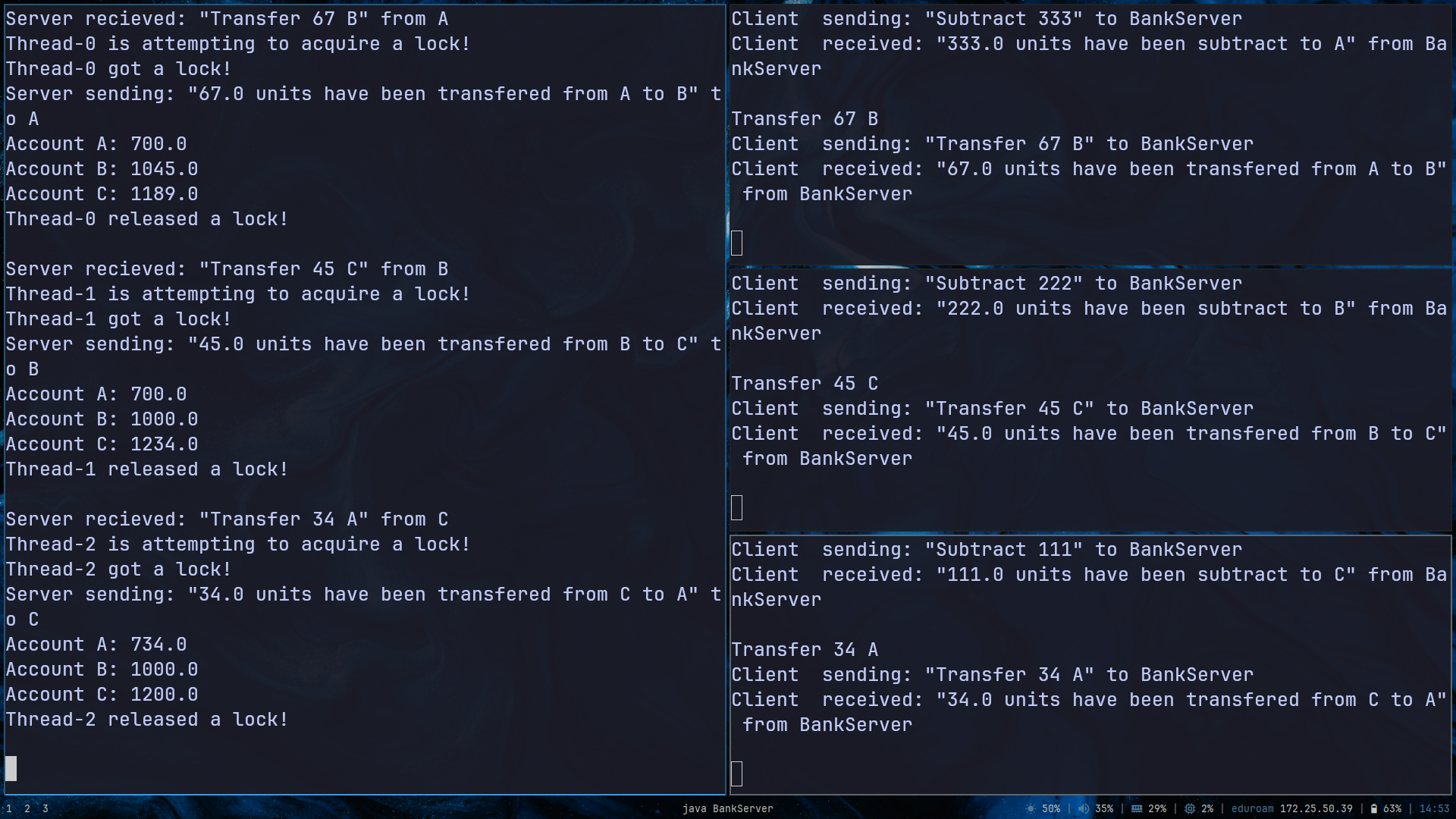
## Adding (add\_money)



## Subtracting (subtract\_money)



## Transferring (transfer\_money)



# Conclusions

In conclusion, I have demonstraded my ability to identify requirements, design, implement and test a network computing application using a concurrent client-server model. All source code can be found in the github repository in the appendix.

# Appendix A

GitHub repo containing code + report: <https://github.com/dlnrd/CS3004-coursework>

Desciptions of classes can be found in README.md: [https://github.com/dlnrd/CS3004-coursework?tab=readme-ov-file#description-of-each-class](https://github.com/dlnrd/CS3004-coursework?tab=readme-ov-file" \l "description-of-each-class)