Table of Contents

1. Introduction

The assignment aims to design and develop the WLFB Bank Application (WLFB), where each user has a client which will communicate to a bank server. The application should be a multi-threaded client-server system where each client can simultaneously add, subtract and transfer money to and from their account. Both the client and server will be developed in a command-line interface (CLI). The server will hold the client bank accounts and execute the operations as per the client's instructions. Additionally, a custom logger feature will be developed that will allow me to debug and demonstrate how the server operates.

2. Requirements

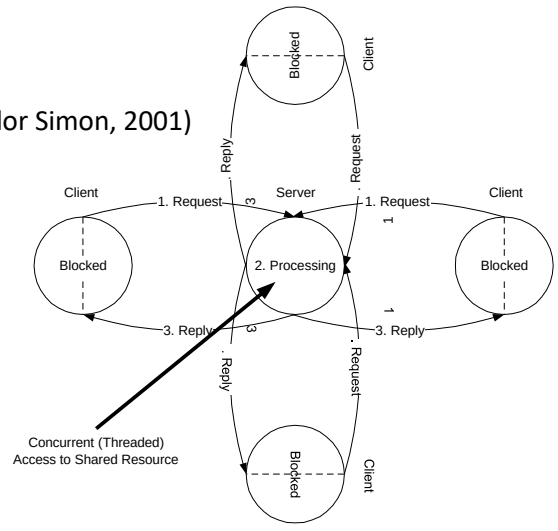
- Create a multi-threaded client-server application that uses locking, which demonstrates how issues in network computing such as concurrency, are dealt with in a multi-client scenario.
- Each client will start at 1000 units (an arbitrary and artificial currency) in their account.
- The account balance can go below zero.
- Implement add (amount) operation - that will add the specified amount to their account.
- Implement subtract (amount) operation - that will subtract the specified amount from their account.
- Implement transfer (senderID, receiverID, amount) operation - that will transfer the specified money from sender to receiver account.

Additional Features:

- Implement an exit operation – that will terminate the client-server connection and stop the client.
- Simple log files system that writes the server operations to a log file (in real-time) – which makes debugging much simpler.
- The clients will be automatically assigned to make the application scalable.
- Implement client input validation to ensure no exceptions are thrown such as validating if the account exists before operating.

3. Design

(Taylor Simon, 2001)



The diagram above shows the architecture that the application is built on, where multiple bank clients can simultaneously communicate with a single bank server. The bank server uses a shared state object which stores the bank information for each client. The bank server implements a strict 2 phase-locking mechanism to handle any data concurrency issues. A lock will ensure that only one client can execute transactions at a time – isolating the shared object and, further verifying that data consistency is preserved. Furthermore, it will prevent any dirty reads and premature writes from occurring and thereby reduce the chance of conflicts between threads attempting to access the object.

The client's information i.e., account name and balance will be stored in a custom shared object called “bank account”. The application is designed to be executed in a command-line interface (CLI), where it will communicate to the server through port/socket and independent threads will execute operations for each client simultaneously. Several clients can connect to the same port at the same time, where the data received is disguised by the client address or by the unique client id.

As the client and server are in a direct relation without any intermediate layer, which provides the fastest response rate and gives the best performance. However, growing clients will affect the performance, and it has the least portability. In this architecture, the client handles the presentation layer (user interface), and the server handles both the business logic and database logic.

The bank server is designed to perform a range of validation on the data received by the bank client such as validating if the account referenced exists or checking if the amount entered only contains integers and lastly if the operation is valid or not. It is crucial to validate inputs, to guarantee that transactions are executed successfully without throwing any exception which might result in deadlock or termination.

Protocol Table

	BankClient		**BankServer**
			[run BankServer on port X and initialise the shared bank state object]
	[run BankClient]		
	[connect to BankServer]		
	WHILE NOT TERMINATED		WHILE NOT TERMINATED
			Wait for BankClient connection
			When BankClient connection is established
			SEND “Initialised Client and IO Connection” TO BankClient
	RECEIVE “Initialised Client and IO Connection” FROM BankServer		
	WHILE NOT TERMINATED		WHILE NOT TERMINATED
	READ userInput FROM console		
	SEND userInput TO BankServer		
			RECEIVE userInput FROM BankClient
			[Acquire lock on the shared bank state object]
			IF command AND amount AND account IN userInput IS VALID THEN
			IF message = “ADD” THEN
			[ADD amount to account]
			SEND “[amount] added to [account]” TO BankClient
			END IF
			IF message = “SUBTRACT” THEN
			[SUBTRACT amount from account]
			SEND “[amount] subtracted from [account]” TO BankClient
			END IF
			IF message = “TRANSFER” THEN
			[TRANSFER amount from sender account to receiver account]
			SEND “[amount] transferred from [sender] to [receiver]” TO BankClient
			END IF
			IF message = “EXIT” THEN
			SEND “Bye, see you soon!” TO BankClient
			[TERMINATE BankClient connection]
			END IF
			ELSE
			SEND “Invalid request received - try (add, subtract, transfer, exit)” TO BankClient
			END IF
			[Release lock on the shared bank state object]
	RECEIVE message FROM BankServer		
	IF message = “Bye, see you soon!” THEN		
	[Connection terminated]		
	ELSE		
	[Print message to the console]		
	END WHILE		END WHILE
	[Terminate BankClient]		[Terminate BankServer]
	END WHILE		END WHILE

4. Implementation

- Acquire Lock:

```
public synchronized void acquireLock() throws InterruptedException {
    Thread currentThread = Thread.currentThread();
    System.out.println(getTS() + currentThread.getName() + ": attempting to acquire lock");
    threadsWaiting++;
    while(accessing) {
        System.out.println(getTS() + currentThread.getName()
                + ": waiting to receive lock - inuse (" + threadsWaiting + " waiting)");
        wait();
    }
    threadsWaiting--;
    accessing = true;
    System.out.println(getTS() + currentThread.getName() + ": lock obtained");
}
```

The acquire lock method was implemented using the synchronized keyword, which will provide synchronization by having an internal monitor lock as without it thread interference and memory consistency errors would originate. When the method is called, clientID is retrieved from the threads name. If the object is not already locked then the current thread locks the object (by setting accessing to true) and then it is permitted to operate on it. If the object is being accessed, then the thread must wait until it is unlocked again. The variable “threadsWaiting”, underlines the number of threads waiting in the queue to acquire the lock.

- Release Lock:

```
public synchronized void releaseLock() {
    accessing = false;
    notifyAll();
    Thread currentThread = Thread.currentThread();
    System.out.println(getTS() + currentThread.getName() + ": lock released");
}
```

Once again the method is using a synchronized keyword, to ensure that java’s internal monitor lock preserves synchronization. The release lock method is executed once the current thread - with the lock, has finished executing and the lock is released by setting the variable “accessing” to false. The method wakes up all the threads that had called the method “wait()” by executing “notifyAll()”. Lastly, a print statement is displayed to the console with the timestamp and the thread’s name.

- Threads Initialisation & Assignment:

```
while(listening) {
    new BankServerThread(BankServerSocket.accept(), sharedBankStateObj).start();
}
```

```
public void run() {
    try{
        String clientUniqueAccountID = sharedBankStateObj.createAccount();
        PrintWriter out = new PrintWriter(bankSocket.getOutputStream(), autoFlush: true);
        BufferedReader in = new BufferedReader(new InputStreamReader(bankSocket.getInputStream()));
        String inputLn, outputLn;
        out.println(clientUniqueAccountID);
        System.out.println(sharedBankStateObj.getTS() + "Initialised Client-ID: " + clientUniqueAccountID);
        Thread current = Thread.currentThread();
        current.setName(clientUniqueAccountID);
```

```
protected String createAccount() {
    BankAccount clientAcc;
    if(clientsList.isEmpty()) {
        clientAcc = new BankAccount( accountNo: "Account1", initialClientBalance);
    } else {
        clientAcc = new BankAccount( accountNo: "Account" + (clientsList.size() + 1), initialClientBalance);
    }
    clientsList.add(clientAcc);
    return clientAcc.getAccountNo();
}
```

The first screenshot demonstrates that when a client connection is established, a new thread is initialised and the “start()” method is executed. The thread has access to the shared bank state object, that holds all client account information. The “start()” method, will execute the “run()” method within the “BankServerThread” class and automatically assign each client with a unique client ID. The “createAccount” method increments the clientID and creates a new bank account with a balance of 1000 units. Lastly, the clientID is set as the threads name - to let the server identify the client.

- Client & Server Socket:

Server

```
ServerSocket BankServerSocket = null;
boolean listening = true;
String BankServerName = "WLFB-BankServer";
int BankServerPortNumber = 4243;
double initialClientBalance = 1000;

SharedBankState sharedBankStateObj = new SharedBankState(initialClientBalance);
```

Server

```
try {
    BankServerSocket = new ServerSocket(BankServerPortNumber);
} catch (IOException e) {
    System.err.println("Could not start " + BankServerName + " on specified port(" + BankServerPortNumber + ")");
    System.exit( status: -1);
}
```

Client

```
Socket BankClientSocket = null;
PrintWriter out = null;
BufferedReader in = null;
int BankSocketNumber = 4243;
String BankServerHost = "localhost";
String BankClientID;

try{
    BankClientSocket = new Socket(BankServerHost, BankSocketNumber);
    out = new PrintWriter(BankClientSocket.getOutputStream(), autoFlush: true);
    in = new BufferedReader(new InputStreamReader(BankClientSocket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: " + BankServerHost);
    System.exit( status: 1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection to: "+ BankSocketNumber);
    System.exit( status: 1);
}
```

The bank server utilises a socket – is an endpoint used to communicate between two machines. In this instance, the bank server will attempt to initialise a new server socket on the given port (4243). If the socket is created successfully then it will wait for any client connection to be established otherwise, it will raise an I/O exception and terminate the server. Similarly, the client will attempt to initialise a new socket and try to establish a connection to the bank server host on the specified port. If the specified is not found then an unknown host exception is thrown. Both the client and server open a PrintWriter and a BufferedReader on the socket, where data is sent and received between the two machines. Lastly, a shared object is also created with each client having a balance of 1000 units.

- Acquiring & Releasing Lock Procedure:

```
while((inputLn = in.readLine()) != null) {
    try {
        sharedBankStateObj.acquireLock();
        outputLn = sharedBankStateObj.processInputs(inputLn);
        out.println(outputLn);
        if(inputLn.equalsIgnoreCase( anotherString: "exit")) {
            sharedBankStateObj.releaseLock();
            break;
        }
        sharedBankStateObj.releaseLock();
    } catch (InterruptedException e) {
        System.err.println("Failed to get lock when reading: " + e);
    }
}
```

The screenshot demonstrates an implementation of the strictly 2 phase-locking mechanisms. Firstly, the data is received from the PrintWriter bound to the socket. Before the received instructions from the client can be executed, the thread must acquire the lock by calling the “acquireLock()”. Once the thread is permitted to read/write on the shared object, it will execute the operations by calling the “processInputs()”. The output is then returned to the BufferedReader bound to the socket and a simple conditional statement validates if an exit response is issued, if so then the client-server connection will be terminated and the thread stops. The lock will be released either if the task is completed or if the thread will stop, to ensure the server does not halt in a deadlock – where two or more threads wait forever for a lock that is obtained by another thread resulting in a halt.

- Add, Subtract, Transfer & Exit Operations:

```
protected void addMoney(String accountNo, double moneyToAdd) {
    boolean accountFound = false;
    if(!clientsList.isEmpty()) {
        for (BankAccount current : clientsList) {
            if(current.getAccountNo().equalsIgnoreCase(accountNo)) {
                double originalBalance = current.getBalance();
                current.setBalance(current.getBalance() + Math.abs(moneyToAdd));
                System.out.println(getTS() + "(ADD) " + "Original Balance: £" + originalBalance
                        + ", Updated Balance: £" + current.getBalance());
                accountFound = true;
                break;
            }
        }
        if(!accountFound) {
            System.err.println("(ERROR) Account not found in system");
        }
    } else {
        System.err.println("(ERROR) Bank has no accounts");
    }
}
```

ADD

```
protected void subtractMoney(String accountNo, double moneyToRemove) {
    boolean accountFound = false;
    if(!clientsList.isEmpty()) {
        for (BankAccount current : clientsList) {
            if(current.getAccountNo().equalsIgnoreCase(accountNo)) {
                double originalBalance = current.getBalance();
                current.setBalance(current.getBalance() - Math.abs(moneyToRemove));
                System.out.println(getTS() + "(SUBTRACT) " + "Original Balance: £" + originalBalance
                        + ", Updated Balance: £" + current.getBalance());
                accountFound = true;
                break;
            }
        }
        if(!accountFound) {
            System.err.println("(ERROR) Account not found in system");
        }
    } else {
        System.err.println("(ERROR) Bank has no accounts");
    }
}
```

SUBTRACT

```
protected void transferMoney(String sender, String receiver, double amount) {
    if(!clientsList.isEmpty()) {
        BankAccount senderBankAcc = null, receiverBankAcc = null;

        for (BankAccount current : clientsList) {
            if(current.getAccountNo().equalsIgnoreCase(sender))
                senderBankAcc = current;
            else if (current.getAccountNo().equalsIgnoreCase(receiver))
                receiverBankAcc = current;
        }

        if(senderBankAcc != null && receiverBankAcc != null) {
            double originalSenderBal = senderBankAcc.getBalance(), originalReceiverBal = receiverBankAcc.getBalance();
            senderBankAcc.setBalance(senderBankAcc.getBalance() - Math.abs(amount));
            System.out.println(getTS() + "(TRANSFER) "+ "Original Balance: £" + originalSenderBal
                    + ", Updated Sender Balance: £" + senderBankAcc.getBalance());
            receiverBankAcc.setBalance(receiverBankAcc.getBalance() + Math.abs(amount));
            System.out.println(getTS() + "(TRANSFER) "+ "Original Balance: £" + originalReceiverBal
                    + ", Updated Receiver Balance: £" + receiverBankAcc.getBalance());
        }
    } else {
        System.err.println("(ERROR) Bank has no accounts");
    }
}
```

TRANSFER

```
} else if(command.equalsIgnoreCase( anotherString: "exit")) {
    System.out.println(getTS() + "(EXIT) Client Connection Terminated");
    theOutput = "Bye, see you soon! :)";
```

EXIT

Firstly, all operations undergo a range of checks to validate the amount and account number. The amount is validated to check it only contains numbers and no alphabets or symbols. The account number is checked and retrieved as a bank account object else, an error would be thrown. Each method uses the absolute function provided to ensure that negative numbers are not used at any stage. If the exit operation is performed then the client connection is terminated.

- Custom File Logger (Additional Feature):

```
public class CustomMultiOutputLogger extends OutputStream {

    OutputStream[] outputs;

    public CustomMultiOutputLogger(OutputStream... streams) { this.outputs = streams; }

    @Override
    public void write(int b) throws IOException {
        for (OutputStream out : outputs) {
            out.write(b);
        }
    }

    @Override
    public void write(byte[] b) throws IOException {
        for (OutputStream out : outputs) {
            out.write(b);
        }
    }

    @Override
    public void write(byte[] b, int off, int len) throws IOException {
        for (OutputStream out : outputs) {
            out.write(b, off, len);
        }
    }
}
```

```
FileOutputStream file = new FileOutputStream( name: "ServerLogs.txt",  append: true);
CustomMultiOutputLogger stdOut = new CustomMultiOutputLogger(System.out, file);
CustomMultiOutputLogger stdErr = new CustomMultiOutputLogger(System.err, file);
PrintStream stdOutStream = new PrintStream(stdOut);
PrintStream stdErrStream = new PrintStream(stdErr);
System.setOut(stdOutStream);
System.setErr(stdErrStream);
```

```
[2021/11/30.17:34:58.424] WLFB-BankServer started
[2021/11/30.17:35:02.515] Initialised Client-ID: Account1
[2021/11/30.17:35:05.054] Initialised Client-ID: Account2
[2021/11/30.17:35:07.706] Account1: attempting to acquire lock
[2021/11/30.17:35:07.707] Account1: lock obtained
[2021/11/30.17:35:07.711] (ADD) Original Balance: £1000.0, Updated Balance: £1050.0
[2021/11/30.17:35:07.734] Server: £50.0 added to Account1
[2021/11/30.17:35:07.736] Bank Accounts:
Account1(£1050.0), Account2(£1000.0),
[2021/11/30.17:35:07.740] Account1: lock released
java.net.SocketException: Connection reset
    at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:323)
    at java.base/sun.nio.ch.NioSocketImpl.read(NioSocketImpl.java:350)
    at java.base/sun.nio.ch.NioSocketImpl$1.read(NioSocketImpl.java:803)
    at java.base/java.net.Socket$SocketInputStream.read(Socket.java:966)
    at java.base/sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:270)
    at java.base/sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:313)
```

LOG Text File

The screenshots illustrate the implementation of a custom file logger, which is designed to display the outputs to the application's console and write to a text file in real-time. This design will further help to track and log transactions that occur on the server and improve debugging support that is available to developers. The design is configured such that the standard output and standard error are redirected to be displayed to the console and written/appended by the FileOutputStream to a text file.

5. Testing

- **Add Money:**

Client A

Server-End

```
[2021/11/30.18:15:35.448] WLFB-BankServer started
[2021/11/30.18:15:39.020] Initialised Client-ID: Account1
[2021/11/30.18:15:41.897] Initialised Client-ID: Account2
[2021/11/30.18:15:45.941] Initialised Client-ID: Account3
[2021/11/30.18:16:14.292] Account1: attempting to acquire lock
[2021/11/30.18:16:14.293] Account1: lock obtained
[2021/11/30.18:16:14.300] (ADD) Original Balance: £1000.0, Updated Balance: £1250.0
[2021/11/30.18:16:14.321] Server: £250.0 added to Account1
[2021/11/30.18:16:14.322] Bank Accounts:
Account1(£1250.0), Account2(£1000.0), Account3(£1000.0),
[2021/11/30.18:16:14.328] Account1: lock released
```

Client-End

```
Initialised Account1 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
add 250
Sending: add 250
Received: £250.0 added to Account1
```

Client B

Server-End

```
[2021/11/30.18:16:25.295] Account2: attempting to acquire lock
[2021/11/30.18:16:25.296] Account2: lock obtained
[2021/11/30.18:16:25.297] (ADD) Original Balance: £1000.0, Updated Balance: £1420.0
[2021/11/30.18:16:25.300] Server: £420.0 added to Account2
[2021/11/30.18:16:25.301] Bank Accounts:
Account1(£1250.0), Account2(£1420.0), Account3(£1000.0),
[2021/11/30.18:16:25.302] Account2: lock released
```

Client-End

```
Initialised Account2 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
add 420
Sending: add 420
Received: £420.0 added to Account2
```

- **Subtract Money:**

Client A

Server-End

```
[2021/11/30.18:18:58.715] WLFB-BankServer started
[2021/11/30.18:19:02.967] Initialised Client-ID: Account1
[2021/11/30.18:19:06.768] Initialised Client-ID: Account2
[2021/11/30.18:19:09.812] Initialised Client-ID: Account3
[2021/11/30.18:19:16.028] Account1: attempting to acquire lock
[2021/11/30.18:19:16.029] Account1: lock obtained
[2021/11/30.18:19:16.038] (SUBTRACT) Original Balance: £1000.0, Updated Balance: £901.0
[2021/11/30.18:19:16.062] Server: £99.0 subtracted from Account1
[2021/11/30.18:19:16.063] Bank Accounts:
Account1(£901.0), Account2(£1000.0), Account3(£1000.0),
[2021/11/30.18:19:16.067] Account1: lock released
```

Client-End

```
Initialised Account1 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
subtract 99
Sending: subtract 99
Received: £99.0 subtracted from Account1
```

Client C

Server-End

```
[2021/11/30.18:19:24.023] Account3: attempting to acquire lock
[2021/11/30.18:19:24.024] Account3: lock obtained
[2021/11/30.18:19:24.025] (SUBTRACT) Original Balance: £1000.0, Updated Balance: £431.0
[2021/11/30.18:19:24.026] Server: £569.0 subtracted from Account3
[2021/11/30.18:19:24.026] Bank Accounts:
Account1(£901.0), Account2(£1000.0), Account3(£431.0),
[2021/11/30.18:19:24.027] Account3: lock released
```

Client-End

```
Initialised Account3 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
subtract 569
Sending: subtract 569
Received: £569.0 subtracted from Account3
```

- **Transfer Money:**

Client A

Server-End

```
[2021/11/30.18:22:07.754] WLFB-BankServer started
[2021/11/30.18:22:10.902] Initialised Client-ID: Account1
[2021/11/30.18:22:14.155] Initialised Client-ID: Account2
[2021/11/30.18:22:17.552] Initialised Client-ID: Account3
[2021/11/30.18:22:42.776] Account1: attempting to acquire lock
[2021/11/30.18:22:42.777] Account1: lock obtained
[2021/11/30.18:22:42.783] (TRANSFER) Original Balance: £1000.0, Updated Sender Balance: £275.0
[2021/11/30.18:22:42.799] (TRANSFER) Original Balance: £1000.0, Updated Receiver Balance: £1725.0
[2021/11/30.18:22:42.816] Server: £725.0 transferred from account1 to account2
[2021/11/30.18:22:42.816] Bank Accounts:
Account1(£275.0), Account2(£1725.0), Account3(£1000.0),
[2021/11/30.18:22:42.826] Account1: lock released
```

Client-End

```
Initialised Account1 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
transfer account1 account2 725
Sending: transfer account1 account2 725
Received: £725.0 transferred from account1 to account2
```

Client B

Server-End

```
[2021/11/30.18:23:02.996] Account2: attempting to acquire lock
[2021/11/30.18:23:02.998] Account2: lock obtained
[2021/11/30.18:23:03.000] (TRANSFER) Original Balance: £1725.0, Updated Sender Balance: £1625.0
[2021/11/30.18:23:03.000] (TRANSFER) Original Balance: £1000.0, Updated Receiver Balance: £1100.0
[2021/11/30.18:23:03.002] Server: £100.0 transferred from account2 to account3
[2021/11/30.18:23:03.003] Bank Accounts:
Account1(£275.0), Account2(£1625.0), Account3(£1100.0),
[2021/11/30.18:23:03.005] Account2: lock released
```

Client-End

```
Initialised Account2 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
transfer account2 account3 100
Sending: transfer account2 account3 100
Received: £100.0 transferred from account2 to account3
```

- **Exit Operation:**

Client A

Server-End

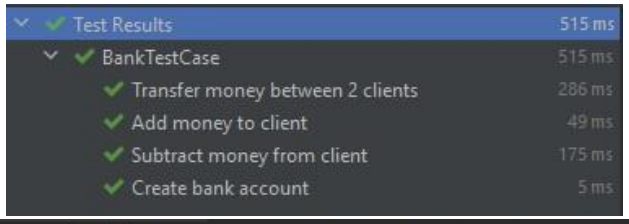
```
[2021/11/30.18:35:57.626] WLFB-BankServer started
[2021/11/30.18:36:16.424] Initialised Client-ID: Account1
[2021/11/30.18:36:19.692] Initialised Client-ID: Account2
[2021/11/30.18:36:26.104] Account1: attempting to acquire lock
[2021/11/30.18:36:26.105] Account1: lock obtained
[2021/11/30.18:36:26.106] (EXIT) Client Connection Terminated
[2021/11/30.18:36:26.110] Server: Bye, see you soon! :)
[2021/11/30.18:36:26.111] Bank Accounts:
Account1(£1000.0), Account2(£1000.0),
[2021/11/30.18:36:26.116] Account1: lock released
```

Client-End

```
Initialised Account1 client and IO connections
Commands:
-> add amount
-> subtract amount
-> transfer senderID receiverID amount
-> exit
exit
Sending: exit
Received: Bye, see you soon! :)

Process finished with exit code 0
```

- **Additional Testcases (100 runs per subtest):**




```
[2021/11/30.18:55:43.333] (TRANSFER) Original Balance: £1000.0, Updated Sender Balance: £927.2436319967132
[2021/11/30.18:55:43.361] (TRANSFER) Original Balance: £1000.0, Updated Receiver Balance: £1072.7563680032868
[2021/11/30.18:55:43.368] (TRANSFER) Original Balance: £927.2436319967132, Updated Sender Balance: £858.9212848207287
[2021/11/30.18:55:43.368] (TRANSFER) Original Balance: £1072.7563680032868, Updated Receiver Balance: £1141.0787151792713
[2021/11/30.18:55:43.372] (TRANSFER) Original Balance: £858.9212848207287, Updated Sender Balance: £828.0493392874628
[2021/11/30.18:55:43.374] (TRANSFER) Original Balance: £1141.0787151792713, Updated Receiver Balance: £1171.9506607125372
[2021/11/30.18:55:43.375] (TRANSFER) Original Balance: £828.0493392874628, Updated Sender Balance: £800.3414902800491
```

```
[2021/11/30.18:55:43.499] (ADD) Original Balance: £1000.0, Updated Balance: £1072.7563680032868
[2021/11/30.18:55:43.506] (ADD) Original Balance: £1072.7563680032868, Updated Balance: £1141.0787151792713
[2021/11/30.18:55:43.506] (ADD) Original Balance: £1141.0787151792713, Updated Balance: £1171.9506607125372
[2021/11/30.18:55:43.506] (ADD) Original Balance: £1171.9506607125372, Updated Balance: £1199.6585097199509
[2021/11/30.18:55:43.506] (ADD) Original Balance: £1199.6585097199509, Updated Balance: £1266.2134048994083
[2021/11/30.18:55:43.507] (ADD) Original Balance: £1266.2134048994083, Updated Balance: £1356.550631366626
```

```
[2021/11/30.18:55:43.555] (SUBTRACT) Original Balance: £1000.0, Updated Balance: £927.2436319967132
[2021/11/30.18:55:43.558] (SUBTRACT) Original Balance: £927.2436319967132, Updated Balance: £858.9212848207287
[2021/11/30.18:55:43.559] (SUBTRACT) Original Balance: £858.9212848207287, Updated Balance: £828.0493392874628
[2021/11/30.18:55:43.559] (SUBTRACT) Original Balance: £828.0493392874628, Updated Balance: £800.3414902800491
[2021/11/30.18:55:43.560] (SUBTRACT) Original Balance: £800.3414902800491, Updated Balance: £733.7865951005917
[2021/11/30.18:55:43.560] (SUBTRACT) Original Balance: £733.7865951005917, Updated Balance: £643.449368633374
```

6. Conclusion

In conclusion, all the requirements listed were successfully implemented which demonstrated my understanding of advanced network computing. The application was critically evaluated and broken down into workable requirements. Additional features like a custom file logger, an “exit” operation and input validations were also developed to further improve the system’s reliability, usability and maintainability. Finally, the bank application demonstrated how concurrency issues in network computing are dealt with in a multi-threaded client-server system.

7. Reference

Taylor Simon. (2001). *CS3004 Session 2 Design and Implementation I, lecture notes, CS3004 Network Computing (A 2021/2), Brunel University London*. https://blackboard.brunel.ac.uk/bbcswebdav/pid-1478498-dt-content-rid-6952625_1/xid6952625_1

Appendix A: Code

The code will be included (ZIP file) with the assignment submission.

List of classes:

1. BankAccount.java – a custom bank account object to hold client information.
2. BankClient.java – client-end application with a command-line user interface.
3. BankServer.java – server-end application.
4. BankServerThread.java – server thread that is executed for each client connection.
5. SharedBankState.java – server states with all business logic (operations, bank account methods and validations).
6. CustomMultiOutputLogger.java – custom file logger that writes system standard output to a log text file.
7. BankTestCase.java – additional test cases which test core requirements.