Introduction:

This assignment briefly summarises the context of our Banking Application problem in which we will extrapolate our requirements. The report then discusses the design of our system (architecture and what classes will be involved) which will then be highlighted by our implementation of our system.

Requirements:

Our application should consist of a server that will hold the variables (more specifically the bank balance of each account: A, B, and C) as well as performing most of the processing as specified by the Scenario. The server should perform 3 main transactions, adding money to an account, subtracting money from an account or transferring money from one account to another. The context also requires us to create 3 distinct client objects to be used by each user.

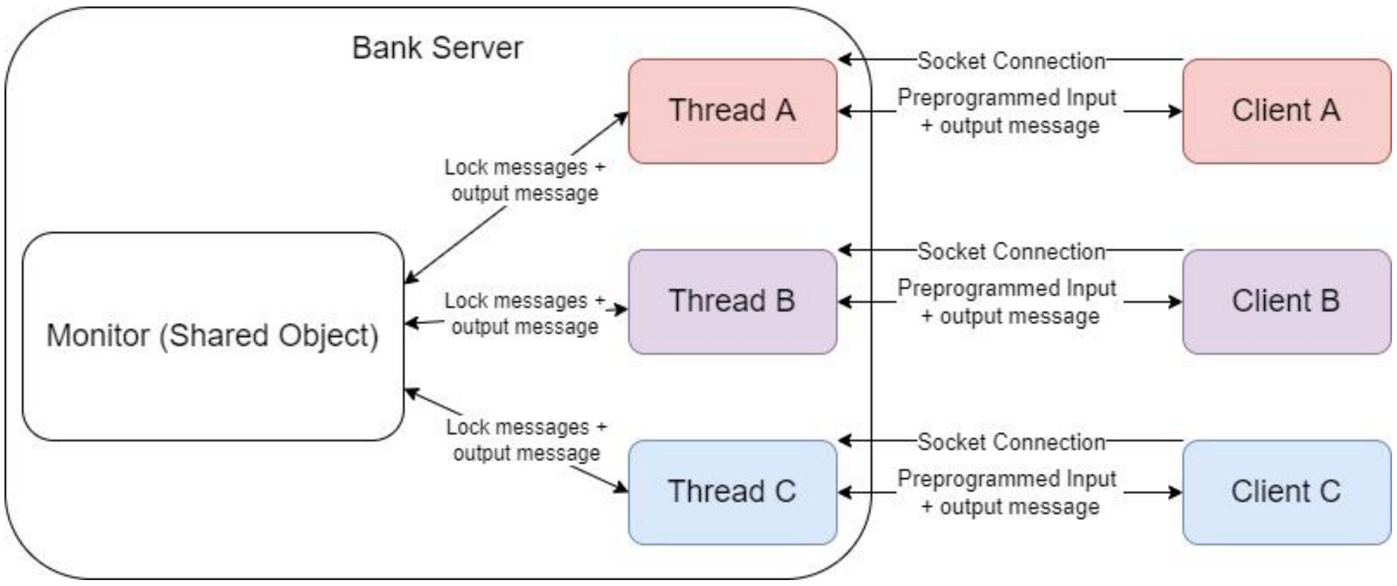
The specification also states that the users are very busy and that there is a constant flux of transactions and messages between the clients and the server. Therefore, it may be applicable to create a multithreaded server instead of a singular heavyweight thread. There are a few benefits to this, which are as follows:

- Design Simplicity (especially for a server-multiclient architecture)
- Concurrency Implications (multicore architecture allows the server to perform other tasks without being blocked in real-life contexts)
- Thread communication through Synchronisation (contributing to the ACID properties of our ‘database’ by monitor sharing)
- And more

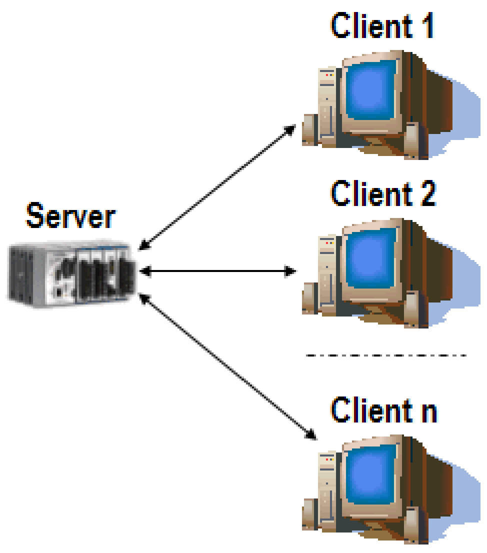
From here, we can conclude that we require a server class, a thread class, a monitor class, and some client classes. Some type of socket connection will be required by the client to connect the client to the bank server.

With the client, we could possibly create two distinct types of clients. We could create a client with preprogramed inputs or one that takes input from the user via the Eclipse console, which will be decided in our design.

Design:


Bank Server
Monitor (Shared Object)
Lock messages + output message
Lock messages + output message
Lock messages + output message
Thread A
Thread B
Thread C
Socket Connection
Preprogrammed Input + output message
Socket Connection
Preprogrammed Input + output message
Socket Connection
Preprogrammed Input + output message
Client A
Client B
Client C

Above is the architecture of how the bank server contains the monitor, which are inextricably linked to each singular thread which are connected to each distinct client. They also describe the communication processes between each class. It will follow a multi-client server pattern.



Atomicity

As the program consists of many transactions, our functions will need to be atomic – they perform all or nothing, and so validity checks will help us in this procedure. For instance, in the transfer money function, we may end up subtracting money from a client account only to find out that the recipient account "F" does not exist, and the add money portion of the Transfer transaction does not go through. Hence, it is important to develop the Transfer function without incorporating the add money function or subtract money function.

The Bank Server Threads: Thread A, Thread B and Thread C

The bank server threads take in 3 parameters for their creation – A socket (which will be an accepted socket to the Bank Server), the shared object monitor and the bank server thread name. The bank server thread will have an input stream from a buffered reader (which will be the output message from the client) and an output stream into a print writer (which will be the input message to the client). For the sake of convenience, a function called Critical_Section (which includes lock acquisition and release) will be added. The thread should constantly listen for an input stream from a client until the socket is closed.

Three of these threads will be created in the Bank Server file, each corresponding to a distinct client.

Consistency?

Consistency could be checked relatively easily by performing validity checks after a transaction occurs and to revert them if any invalid transformations occur. However due to the time limitations (and the tediousness of writing protocol tables), this will be omitted in our design and implementation.

Isolation - The Monitor: Balance A, Balance B, Balance C …

The three variables will be held in a monitor created in the server and shared by all three threads created by the Bank Server: thread A, thread B and thread C. We will also have another variable held by this monitor called Accessing that determines if the monitor is in use (in which case, a thread will wait until it is notified) or not in use (in which case a lock is acquired by some thread or the thread with the highest priority). Sharing resources through mutual exclusion locks can prevent the common problem of deadlocks seen in some applications.

This monitor will have the acquire lock and release lock function that will prevent multiple threads from updating the 3 account variables all at once. The monitor will also be responsible for having an input processor (to process the user command into its constituent parameters), validity check functions to check if the user has entered a sensible input and the transaction functions: Add, Subtract, Transfer.

The input processing function relies on the input message sent by the user **as well as** the name of the thread: A, B or C. Therefore, what messages each thread receives is dependent upon the order that the client objects are run – hence different client object initialisation orders result in different transactions for each variable.

Durability?

Durability is defined as permanent transformations to the databases, even in unexpected scenarios such as a power outage. Our program cannot be defined as durable as the client accounts' balances are always reset to 1000 units each time the Bank Server is initialised. ☹

Class	**Functions / Purposes**
Bank Server	Main – server initialisation, threads initialisation, monitor initialisation, server socket initialisation
Bank Server Thread	Run – initialisation of input and output stream, listen in for incoming messages/input streams, and perform critical section, if necessary, before returning an output stream. Crit_Section – the critical section of the code to be executed once a lock is acquired. Logically speaking, the lock acquisition and release should be outside this function but has been added for the sake of convenience.
Client	Random_Time – creates a random discrete time value between 7000 - 11000. We just want client subclasses to inherit this method.
Client A	Main – server connection initialisation, initialisation of input and output stream. Will then send message “add 100”, receive a message from the server, send message “subtract 500” and then receive a message from the server.
Client B	Main – server connection initialisation, initialisation of input and output stream. Will then send message “add 2000”, receive a message from the server, send message “transfer a 2000” and then receive a message from the server.
Client C	Main – server connection initialisation, initialisation of input and output stream. Will then send message “subtract 30.5”, receive a message from the server, send message “transfer a 500” and then receive a message from the server.

Yellow boxes in protocol table correspond to a message sending function/step, a blue background corresponds to a validity check step and a green background corresponds to a transaction function performed by the server.

Client to Bank Server Thread Protocol			
CX	Client	SX	Bank Server (Thread)
			[run Bank Server]
			Create BankServerThread A, B and C
	[run ClientA / ClientB / ClientC]		
			[accept Client A connection]
	PRINT “ClientId Sending message [message] to Bank Server”		
#C1	SEND message TO BankServerThread Client A’s 1st Message: “add 100” Client A’s 2nd Message: “subtract 500” Client B’s 1st Message: “add 2000” Client B’s 2nd Message: “transfer a 2000” Client C’s 1st Message: “subtract 30.5” Client C’s 2nd Message: “transfer a 500” [Message format: Add 200 → Server will add 200 units to the account the thread is associated with via the client		

	Subtract 150 → Server will subtract 150 units from the account the thread is associated with via the client Transfer B 190 → Server will transfer 190 units from the account the thread is associated with via the client to account B]		
		#S1	WHILE [RECEIVE message FROM Client A] != null
			[acquire lock and execute Process Input function]
			Capitalise message and split message into parameters array according to “ ”
			IF parameters[0] is “ADD” THEN
			IF account is invalid or value is not a positive double THEN
			THROW Exception e
			GO TO #S2
			END IF
			ELSE
			Perform add money function with parameters[1] and name of thread
			OutputMessage = “[value] added to [threadName]”
			GO TO #S3
			END IF
			ELSE IF parameters[1] is “SUBTRACT” THEN
			IF account is invalid or value is not a positive double THEN
			THROW Exception e
			GO TO #S2
			END IF
			ELSE
			Perform subtract money function with parameters[1] and name of thread
			OutputMessage = “[value] subtracted from [threadName]”
			GO TO #S3
			END IF
			IF parameters[0] is “TRANSFER” THEN
			IF account1 is invalid or value is not a positive double or account2 is invalid or account1 equals account2 THEN
			THROW Exception e
			GO TO #S2
			END IF

			ELSE
			Perform transfer money function with parameters[1], parameters[2] and name of thread
			OutputMessage = “[value] transferred from [threadName] to [recipientAccount]”
			GO TO #S3
			END IF
			ELSE
			OutputMessage = “Invalid command entered”
			GO TO #S3
			END IF
		#S2	CATCH Exception e
			OutputMessage = “Exception occurred parsing command”
			GO TO #S3
		#S3	PRINT ALL BALANCES
			SEND OutputMessage TO Client [Message format: 100 added to A → Indicates 100 has been added to A, where 100 and A are interchangeable 270 subtracted from B → Indicates 270 has been subtracted, where 270 and B are interchangeable 50.5 transferred from C to A → Indicates that account C has transferred 50.5 units to account A, where C, A and 50.5 are interchangeable Invalid command → means the users first word were not add, subtract or transfer Exception occurred parsing command → means the users did not enter valid parameters]
			GO TO #S1
			END WHILE
	RECEIVE OutputMessage FROM BankServerThread		
	PRINT “ClientId recieved message [message] from Bank Server”		
	IF more messages to send		
	GO TO #C1		
	ELSE		

	Close socket		
	Terminate		

Implementation:

This code is to create the bank server structure (with the monitor and the threads A, B and C inside the server), located in the Bank Server class.

```
		// Some server details...
		ServerSocket BankServerSocket= null;
		boolean listening = true;
		String BankServerName = "Bank Server";
		int BankServerNumber = 4545;

		// This will be the balance of the individual accounts A B and C
		double InitialClientBalance = 1000;
		// Create our shared object...
		SharedActionState sharedActionStateObject = new
SharedActionState(InitialClientBalance);

		// Initialise Server Socket...
		try {
			BankServerSocket= new ServerSocket(BankServerNumber);
		} catch (IOException e) {
		    System.err.println("Could not start " + BankServerName + " on
specified port: "+BankServerNumber);
		    System.exit(-1);
		}
		    System.out.println(BankServerName + " started on port: "+
BankServerNumber);
		    System.out.println("Welcome to the Bank Server console...");

		    new BankServerThread(BankServerSocket.accept(), "A",
sharedActionStateObject).start();
		    new BankServerThread(BankServerSocket.accept(), "B",
sharedActionStateObject).start();
		    new BankServerThread(BankServerSocket.accept(), "C",
sharedActionStateObject).start();
```

This code relates to the initialisation of our three variables/balances inside our monitor, utilised by all three threads, located in the Shared Action State class.

```
	Private String ThreadName;
	private double BalanceA;
	private double BalanceB;
	private double BalanceC;
	private  oolean accessing=false;
```

```
        // Constructor for our initial balance in each account
        SharedActionState(double InitialBalance) {
                this.BalanceA = InitialBalance;
                this.BalanceB = InitialBalance;
                this.BalanceC = InitialBalance;
        }
```

The following also describe the lock acquisition and release function described in the design, located in the Shared Action State class.

```
        // Lock Acquisition
        public synchronized void acquireLock() throws InterruptedException{
                Thread currentThread = Thread.currentThread(); // Get reference for
current thread
                while (accessing) {
                    wait();
                  }
                  accessing = true;
                  System.out.println(currentThread.getName()+" acquired a lock");
        }


        // Lock Release
        public synchronized void releaseLock() {
          Thread currentThread = Thread.currentThread(); // Get reference for
current thread
              System.out.println(currentThread.getName()+" released a lock");
                accessing = false;
                notifyAll();
        }
```

The process input implementation corresponds to the algorithm described in the protocol table above. Depending on the value of the first word, it will execute an add money function, subtract money function, transfer money function, return an invalid command message or throw an exception. It will also split the message into parameters to be passed into the transaction functions. This is located in the Shared Action State class.

```
// Function to process inputs messages sent by the threads from the client
        public synchronized String processInput(String myThreadName, String
theInput) {
                System.out.println("Lock acquired");
                System.out.println(myThreadName + " received message ["+
theInput+"]");
                String messageOutput = null;
                theInput = theInput.toUpperCase();

                // Check the input of the string and see which operation to perform
                try {
                    String[] parameters = theInput.split(“ “);
                    String command = parameters[0];

                    if (command.equals(“ADD”)) { // Perform add money function
                          double value = Double.parseDouble(parameters[1]);
                          Add_Money(myThreadName, value);
                          messageOutput = value + “ added to “ + myThreadName;
                    }
                    else if (command.equals(“SUBTRACT”)) { // Perform subtract
money function
                          double value = Double.parseDouble(parameters[1]);
                          Subtract_Money(myThreadName, value);
                          messageOutput = value + “ subtracted from “ +
myThreadName;
                    }
                    else if (command.equals(“TRANSFER”)) { // Perform transfer
money function
                          String recipientAccount = parameters[1];
                          double value = Double.parseDouble(parameters[2]);
                          Transfer_Money(myThreadName, recipientAccount, value);
                          messageOutput = value + “ transferred from “ +
myThreadName + “ to “ + recipientAccount;
                    }
                    else
                          messageOutput = “Invalid command entered”; // Invalid
first command

              } catch (Exception e) {
                    messageOutput = “Exception occured parsing command”; //
Usually occurs for invalid parameters of correct commands
              }

              Print_Balances();
              return messageOutput;
       }
```

The following code snippets describe the 3 different types of validity checks the program performs before each transfer function occurs (inside our protocol tables, these are highlighted in light-blue boxes). All can be found in the Shared Action State class.

```
       Public void Print_Balances() {
              System.out.println(“A Balance: “ + BalanceA);
              System.out.println(“B Balance: “ + BalanceB);
              System.out.println(“C Balance: “ + BalanceC);
       }

       public  oolean Valid_Account_Check(String account) {
              // Validity check…
              ArrayList<String> AccountList = new ArrayList<String>();
              AccountList.add(“A”);
```

```
            AccountList.add("B");
            AccountList.add("C");

            if (AccountList.contains(account))
                  return true;
            else
                  return false;

      }

      public  oolean Positive_Value_Check(double value) {
            if (value > 0)
                  return true;
            else
                  return false;
      }

      public  oolean Transfer_Account_Check(String account1, String account2) {
            if (account1.equals(account2))
                  return false;
            else
                  return true;
      }
```

The following are the examples of these validity checks used inside the transfer function. The next one is used in both the add money and subtract money function to ensure an all or nothing procedure.

```
            // Validity checks to ensure atomicity
            boolean validAccount1 = Valid_Account_Check(account1);
            boolean validAccount2 = Valid_Account_Check(account2);
            boolean transferAccount = Transfer_Account_Check(account1,
account2);
            boolean positiveValue = Positive_Value_Check(value);


            // If checks fail, do not do operation
            if (!(validAccount1 && validAccount2 && transferAccount &&
positiveValue)) {
                  System.out.println("Invalid values in transfer command.
Transfer transaction did not go through.\n");
                  throw new Exception("Invalid values");
            }
```

The next one describes the validity and all or nothing code block for the transfer money function, that checks for positive value, that both accounts are valid and the money isn't being transferred to the same client.

```
      // Validity checks to ensure atomicity
            boolean validAccount1 = Valid_Account_Check(account1);
            boolean validAccount2 = Valid_Account_Check(account2);
            boolean transferAccount = Transfer_Account_Check(account1, 
account2);
            boolean positiveValue = Positive_Value_Check(value);


            // If checks fail, do not do operation
            if (!(validAccount1 && validAccount2 && transferAccount && 
positiveValue)) {
                  System.out.println("Invalid values in transfer command. 
Transfer transaction did not go through.\n");
                  throw new Exception("Invalid values");
            }
            else {
```

All the transfer functions the server performs in the protocol table will be highlighted in light green. The following is how the add money function interacts with one of the three variables, all of which are found in the Shared Action State class.

```
      // Add Money Function
      public void Add_Money(String account, double value) throws Exception {

            …
            else {
            // Check which account to add to and do accordingly...
                  if (account.equals("A"))
                        BalanceA = BalanceA + value;
                  else if (account.equals("B"))
                        BalanceB = BalanceB + value;
                  else if (account.equals("C"))
                        BalanceC = BalanceC + value;

                  // Logging the add_money action
                  System.out.println("Added "+value+" from Account 
"+account+"\n");
            }
      }
```

The following is how the subtract money function interacts with one of the three variables.

```
// Subtract Money Function
      public void Subtract_Money(String account, double value) throws Exception {


            …
            else {
                  // Check which account to subtract from and do accordingly...
                  if (account.equals("A"))
                        BalanceA = BalanceA - value;
                  else if (account.equals("B"))
                        BalanceB = BalanceB - value;
                  else if (account.equals("C"))
                        BalanceC = BalanceC - value;
                  else
                        throw new Exception("");
```

```
                    // Logging the subtract_money action
                    System.out.println("Subtracted "+value+" from Account
"+account+"\n" );
              }
       }
```

The following is how the transfer money function interacts with two of the three variables.

```
public void Transfer_Money(String account1, String account2, double value) throws
Exception {
       ...
              else {
                    // Perform a subtraction to account1...
                    if (account1.equals("A"))
                           BalanceA = BalanceA - value;
                    else if (account1.equals("B"))
                           BalanceB = BalanceB - value;
                    else if (account1.equals("C"))
                           BalanceC = BalanceC - value;

                    // Perform an addition to account2...
                    if (account2.equals("A"))
                           BalanceA = BalanceA + value;
                    else if (account2.equals("B"))
                           BalanceB = BalanceB + value;
                    else if (account2.equals("C"))
                           BalanceC = BalanceC + value;

                    // Logging the transfer money action
                    System.out.println("Transferred "+value+" from Account
"+account1 + " to Account "+account2+"\n");
              }
       }
```

The following is to create a while loop (shown inside the protocol table) within the bank server, or more specifically the bank server thread, that constantly listens for an input given by the associated client. This is located in the Bank Server thread class.

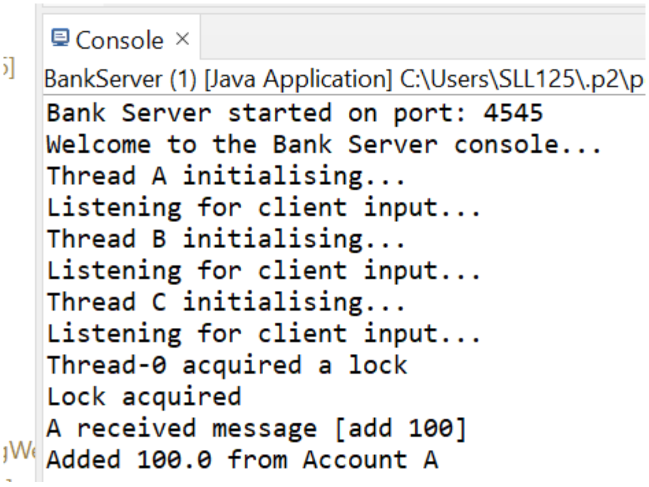
```
System.out.println("Listening for client input...");
                 // Listen for input
                 while ((inputLine = in.readLine()) != null) {
                      try {
                      // Acquire a lock, and then execute the processInput
function, then release the lock...
                      Crit_Section(inputLine, outputLine, out);
                      } catch (Exception e) {
                             System.err.println("Failed to get lock when
reading:"+e);
                      }
                 }
```

Testing:

The following is the preprogramed user input/ message to the server, which should correspond to our screenshots.

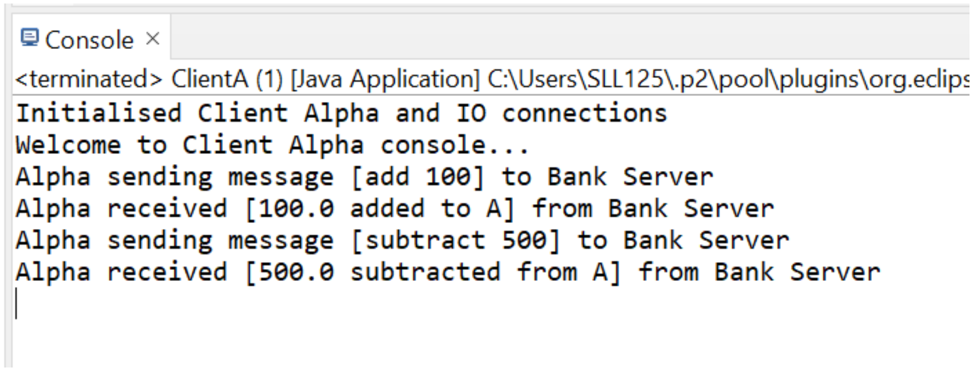
Client	Transaction / Message
Client A	add 100 subtract 500
Client B	add 2000 transfer a 2000
Client C	subtract 30.5 transfer a 500

We can observe successful thread initialisation and socket connection through the Bank Server console, and each respective client. More so, each clients' console contains a message sent output which corresponds to the table above. The received messages also happen to correspond to the message sent by the client in Client A, B and C.



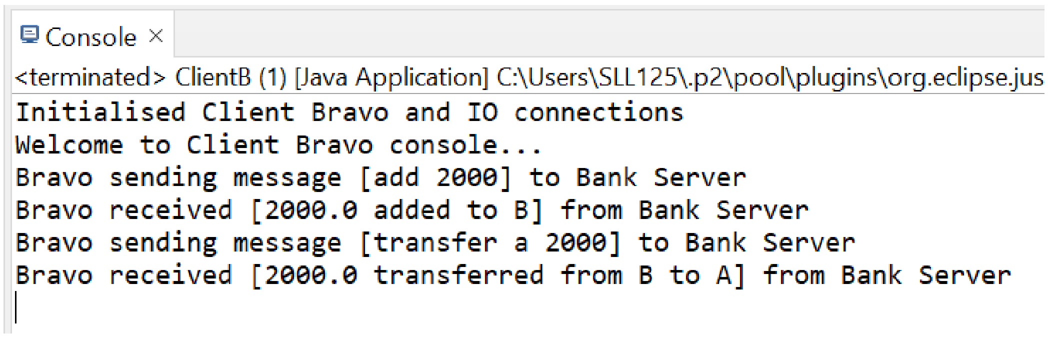
```
Bank Server started on port: 4545
Welcome to the Bank Server console...
Thread A initialising...
Listening for client input...
Thread B initialising...
Listening for client input...
Thread C initialising...
Listening for client input...
Thread-0 acquired a lock
Lock acquired
A received message [add 100]
Added 100.0 from Account A
```

Figure 1: Bank Server Console – proof of thread starts and socket connections


Console ×
<terminated> ClientA (1) [Java Application] C:\Users\SLL125\.p2\pool\plugins\org.eclips

```
Initialised Client Alpha and IO connections
Welcome to Client Alpha console...
Alpha sending message [add 100] to Bank Server
Alpha received [100.0 added to A] from Bank Server
Alpha sending message [subtract 500] to Bank Server
Alpha received [500.0 subtracted from A] from Bank Server
```

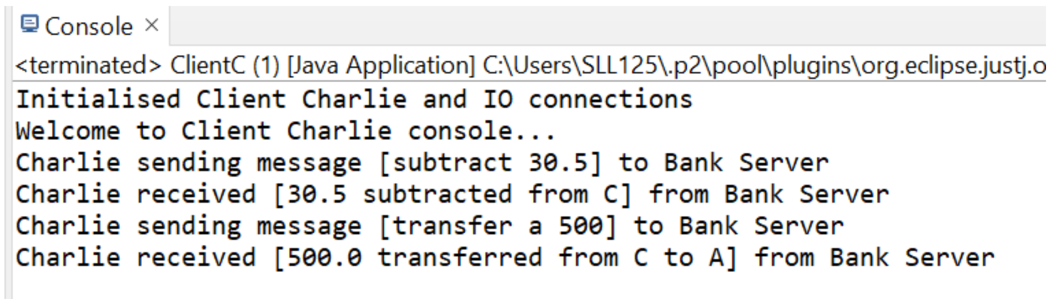
Figure 2: Client A (Alpha) console - proof of transactions


Console
<terminated> ClientB (1) [Java Application] C:\Users\SLL125\.p2\pool\plugins\org.eclipse.jus

```
Initialised Client Bravo and IO connections
Welcome to Client Bravo console...
Bravo sending message [add 2000] to Bank Server
Bravo received [2000.0 added to B] from Bank Server
Bravo sending message [transfer a 2000] to Bank Server
Bravo received [2000.0 transferred from B to A] from Bank Server
```

F

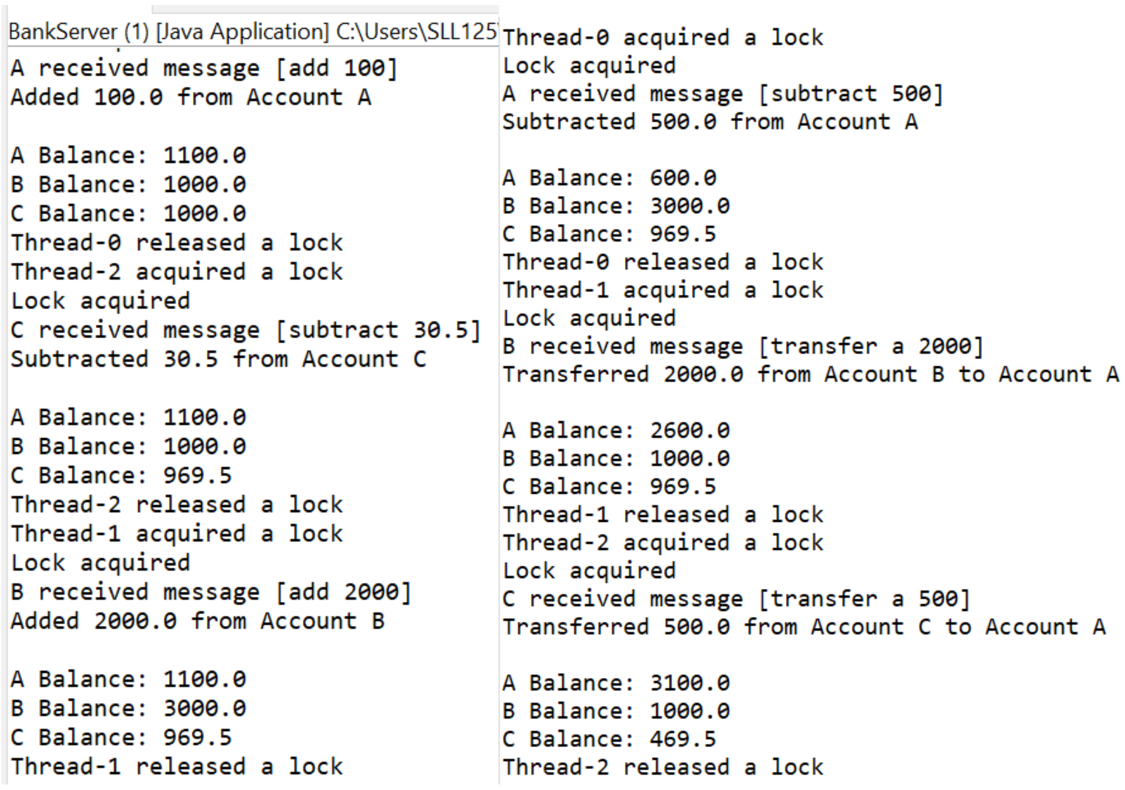
igure 3: Client B (Bravo) - proof of transactions


Console
<terminated> ClientC (1) [Java Application] C:\Users\SLL125\.p2\pool\plugins\org.eclipse.justj.o

```
Initialised Client Charlie and IO connections
Welcome to Client Charlie console...
Charlie sending message [subtract 30.5] to Bank Server
Charlie received [30.5 subtracted from C] from Bank Server
Charlie sending message [transfer a 500] to Bank Server
Charlie received [500.0 transferred from C to A] from Bank Server
```

Figure 4: Client C (Charlie) - proof of transactions

Our program also makes the server/threads print out all the balance variables after each transaction, to ensure that the transactions are correct and do not affect the other accounts.


```
BankServer (1) [Java Application] C:\Users\SLL125
A received message [add 100]
Added 100.0 from Account A

A Balance: 1100.0
B Balance: 1000.0
C Balance: 1000.0
Thread-0 released a lock
Thread-2 acquired a lock
Lock acquired
C received message [subtract 30.5]
Subtracted 30.5 from Account C

A Balance: 1100.0
B Balance: 1000.0
C Balance: 969.5
Thread-2 released a lock
Thread-1 acquired a lock
Lock acquired
B received message [add 2000]
Added 2000.0 from Account B

A Balance: 1100.0
B Balance: 3000.0
C Balance: 969.5
Thread-1 released a lock
```

```
Thread-0 acquired a lock
Lock acquired
A received message [subtract 500]
Subtracted 500.0 from Account A

A Balance: 600.0
B Balance: 3000.0
C Balance: 969.5
Thread-0 released a lock
Thread-1 acquired a lock
Lock acquired
B received message [transfer a 2000]
Transferred 2000.0 from Account B to Account A

A Balance: 2600.0
B Balance: 1000.0
C Balance: 969.5
Thread-1 released a lock
Thread-2 acquired a lock
Lock acquired
C received message [transfer a 500]
Transferred 500.0 from Account C to Account A

A Balance: 3100.0
B Balance: 1000.0
C Balance: 469.5
Thread-2 released a lock
```

Figure 5: Bank Server - output during message exchange. Note that the formatting may be unusual due to a print line. Thread A/B/C will receive a message firstly, perform the necessary transaction, prints a line before displaying the balances of the three accounts. The locking messages may be ignored as the threads are identified by their actual system name rather than the string assigned one we have given.

As the message input correspond to all the bank balances output displayed on the console (like when C transfers A 500, the balances match the transformation expected from the pre-processing state and the postprocessing state)

Conclusion:

We have created a multi-client server application with a multithreading implementation. All requirements have been met to a necessary standard, but considering a larger context, some form of thread initialisation process for a client should be made. There should also be an implementation such that the executions of the client should not matter – that is we should have a thread dedicated to listening to connection requests and creating separate bank server threads if necessary. This way, we may circumvent the above issue of client execution order.

Formatted Code:

BankServer - the class that will be run first, and initialises the shared action state monitor and the bank server threads. It also initialises a server socket that is used by all threads, that is open on port 4545.

```
import java.io.IOException;
import java.net.ServerSocket;

public class BankServer {

      public static void main(String[] args) throws IOException {
              // Some server details...
              ServerSocket BankServerSocket= null;
              boolean listening = true;
              String BankServerName = "Bank Server";
              int BankServerNumber = 4545;

              // This will be the balance of the individual accounts A B and C
              double InitialClientBalance = 1000;
              // Create our shared object...
              SharedActionState sharedActionStateObject = new
SharedActionState(InitialClientBalance);

              // Initialise Server Socket...
              try {
                    BankServerSocket= new ServerSocket(BankServerNumber);
              } catch (IOException e) {
                  System.err.println("Could not start " + BankServerName + " on
specified port: "+BankServerNumber);
                  System.exit(-1);
              }
                  System.out.println(BankServerName + " started on port: "+
BankServerNumber);
                  System.out.println("Welcome to the Bank Server console...");

                    // Let us create a thread for each of our respective
clients...
                  // I believe the threads have to be in the server...

                  new BankServerThread(BankServerSocket.accept(), "A",
sharedActionStateObject).start();
                  new BankServerThread(BankServerSocket.accept(), "B",
sharedActionStateObject).start();
                  new BankServerThread(BankServerSocket.accept(), "C",
sharedActionStateObject).start();


                  try {
                    while (listening){

                          }
                  } catch (Exception e) {
                    System.out.println("Done listening mate");
                  }

              // End the server...
              // BankServerSocket.close();

        }

}
```

SharedActionState – the class where one instance is initialised inside the server. Handles lock acquisition and release, as well as processing inputs from clients, checking validity of input message, edits the three balances that are considered as shared resources and produces an output message for a server to send back to the client.

```
import java.util.ArrayList;

public class SharedActionState {

	private String ThreadName;
	private double BalanceA;
	private double BalanceB;
	private double BalanceC;
	private boolean accessing=false;

	// Constructor for our initial balance in each account
	SharedActionState(double InitialBalance) {
		this.BalanceA = InitialBalance;
		this.BalanceB = InitialBalance;
		this.BalanceC = InitialBalance;
	}


	// Lock Acquisition
	public synchronized void acquireLock() throws InterruptedException{
		   Thread currentThread = Thread.currentThread(); // Get reference for 
current thread
		   while (accessing) {
		        wait();
		      }
		      accessing = true;
		      System.out.println(currentThread.getName()+" acquired a lock");
	}


	// Lock Release
	public synchronized void releaseLock() {
	  Thread currentThread = Thread.currentThread(); // Get reference for 
current thread
	     System.out.println(currentThread.getName()+" released a lock");
		accessing = false;
		notifyAll();
	}

	// Function to process inputs messages sent by the threads from the client
	public synchronized String processInput(String myThreadName, String 
theInput) {
		System.out.println("Lock acquired");
		System.out.println(myThreadName + " received message ["+ 
theInput+"]");
		String messageOutput = null;
		theInput = theInput.toUpperCase();

		// Check the input of the string and see which operation to perform
		try {

			String[] parameters = theInput.split(" ");
```

```
                    String command = parameters[0];

                    if (command.equals("ADD")) { // Perform add money function
                          double value = Double.parseDouble(parameters[1]);
                          Add_Money(myThreadName, value);
                          messageOutput = value + " added to " + myThreadName;
                    }
                    else if (command.equals("SUBTRACT")) { // Perform subtract
money function
                          double value = Double.parseDouble(parameters[1]);
                          Subtract_Money(myThreadName, value);
                          messageOutput = value + " subtracted from " +
myThreadName;
                    }
                    else if (command.equals("TRANSFER")) { // Perform transfer
money function
                          String recipientAccount = parameters[1];
                          double value = Double.parseDouble(parameters[2]);
                          Transfer_Money(myThreadName, recipientAccount, value);
                          messageOutput = value + " transferred from " +
myThreadName + " to " + recipientAccount;
                    }
                    else
                          messageOutput = "Invalid command entered"; // Invalid
first command

             } catch (Exception e) {
                    messageOutput = "Exception occured parsing command"; //
Usually occurs for invalid parameters of correct commands
             }

             Print_Balances();
             return messageOutput;
       }



       // Add Money Function
       public void Add_Money(String account, double value) throws Exception {

             // Validity checks to ensure atomicity
             boolean validAccount = Valid_Account_Check(account);
             boolean positiveValue = Positive_Value_Check(value);

             // If checks fail, do not do operation
             if (!(validAccount && positiveValue)) {
                    System.out.println("Invalid values added. Addition transaction
did not go through.\n");
                    throw new Exception("Invalid values");
             }
             else {
             // Check which account to add to and do accordingly...
                    if (account.equals("A"))
                          BalanceA = BalanceA + value;
                    else if (account.equals("B"))
                          BalanceB = BalanceB + value;
                    else if (account.equals("C"))
                          BalanceC = BalanceC + value;
                    // Logging the add_money action
                    System.out.println("Added "+value+" from Account 
"+account+"\n");
             }
      }


      // Subtract Money Function
      public void Subtract_Money(String account, double value) throws Exception {


             // Validity checks to ensure atomicity
             boolean validAccount = Valid_Account_Check(account);
             boolean positiveValue = Positive_Value_Check(value);

             // If checks fail, do not do operation
             if (!(validAccount && positiveValue)) {
                    System.out.println("Invalid values in subtract command. 
Subtract transaction did not go through.\n");
                    throw new Exception("Invalid values");
             }
             else {
                    // Check which account to subtract from and do accordingly...
                    if (account.equals("A"))
                          BalanceA = BalanceA - value;
                    else if (account.equals("B"))
                          BalanceB = BalanceB - value;
                    else if (account.equals("C"))
                          BalanceC = BalanceC - value;
                    else
                          throw new Exception("");

                    // Logging the subtract_money action
                    System.out.println("Subtracted "+value+" from Account 
"+account+"\n" );
             }
      }


      public void Transfer_Money(String account1, String account2, double value) 
throws Exception {

             // Validity checks to ensure atomicity
             boolean validAccount1 = Valid_Account_Check(account1);
             boolean validAccount2 = Valid_Account_Check(account2);
             boolean transferAccount = Transfer_Account_Check(account1, 
account2);
             boolean positiveValue = Positive_Value_Check(value);


             // If checks fail, do not do operation
             if (!(validAccount1 && validAccount2 && transferAccount && 
positiveValue)) {
                    System.out.println("Invalid values in transfer command. 
Transfer transaction did not go through.\n");
                    throw new Exception("Invalid values");
             }
             else {
                    // Perform a subtraction to account1...
```

```
				if (account1.equals("A"))
					BalanceA = BalanceA - value;
				else if (account1.equals("B"))
					BalanceB = BalanceB - value;
				else if (account1.equals("C"))
					BalanceC = BalanceC - value;

				// Perform an addition to account2...
				if (account2.equals("A"))
					BalanceA = BalanceA + value;
				else if (account2.equals("B"))
					BalanceB = BalanceB + value;
				else if (account2.equals("C"))
					BalanceC = BalanceC + value;

				// Logging the transfer money action
				System.out.println("Transferred "+value+" from Account 
"+account1 + " to Account "+account2+"\n");
			}
	}

	public void Print_Balances() {
		System.out.println("A Balance: " + BalanceA);
		System.out.println("B Balance: " + BalanceB);
		System.out.println("C Balance: " + BalanceC);
	}

	public boolean Valid_Account_Check(String account) {
		// Validity check...
		ArrayList<String> AccountList = new ArrayList<String>();
		AccountList.add("A");
		AccountList.add("B");
		AccountList.add("C");

		if (AccountList.contains(account))
			return true;
		else
			return false;

	}

	public boolean Positive_Value_Check(double value) {
		if (value > 0)
			return true;
		else
			return false;
	}

	public boolean Transfer_Account_Check(String account1, String account2) {
		if (account1.equals(account2))
			return false;
		else
			return true;
	}
}
```

Client – An object for clients to inherit similar methods. Originally planned to optimise all client subclasses by looking for identical blocks of code, and then place them in the client superclass. Contains a function that produce random discrete values of time, allowing the potential execution of threads in different orders for different execution of the application. A minimum of 7000 milliseconds allows me to execute all 3 client objects without any messages being lost.

```
import java.util.Random;

public class Client {

    // Function to get random time values - ensures that the objects arent always
executed in the same sequence.
    // Discrete values allow certain threads to occur at the same time and then
'compete' for the lock
    protected static int Random_Time() {
       int[] random_times = {7000,8000,9000,10000,11000};

       Random rand = new Random();
       int randn = rand.nextInt(5);
       return random_times[randn];

    }

}
```

ClientA – A client responsible for sending preprogramed messages to the server. A is responsible for sending the “add 100” message first, followed by “subtract 500”

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class ClientA extends Client{


       // Constructor from superclass
    public ClientA() {
    }

    // Socket details
    static int ActionSocketNumber = 4545;
    static String ActionServerName = "localhost";
    static String ActionClientID = "Alpha";
    PrintWriter out;
    BufferedReader in;

       public static void main(String[] args) throws IOException,
InterruptedException {
              // Client initiation - run the object
              try {
                     // Initialise the BufferedReader and PrintWriter
                Socket ActionClientSocket = new Socket(ActionServerName,
ActionSocketNumber);
```

```
                       PrintWriter out = new
PrintWriter(ActionClientSocket.getOutputStream(), true);
                       BufferedReader in = new BufferedReader(new
InputStreamReader(ActionClientSocket.getInputStream()));

                    System.out.println("Initialised Client " + ActionClientID + "
and IO connections");
                    System.out.println("Welcome to Client "+ ActionClientID + "
console...");

                       String fromUser = "", fromServer = "";

                    // We can pre-programme some input here for our clients...
                    // Add 100 Transaction
                    fromUser = "add 100";
                    Thread.sleep(Random_Time());
                 System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
                 out.println(fromUser);
                 fromServer = in.readLine();
                 System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

                    // Subtract 500 Transaction
                    fromUser = "subtract 500";
                    Thread.sleep(Random_Time());
                 System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
                 out.println(fromUser);
                 fromServer = in.readLine();
                 System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

                 ActionClientSocket.close();
                 in.close();
                 out.close();

         } catch (UnknownHostException e) {
             System.err.println("Don't know about host: localhost ");
             System.exit(1);
         } catch (IOException e) {
             System.err.println("Couldn't get I/O for the connection to: "+
ActionSocketNumber);
             System.exit(1);
         }

     }

}
```

ClientB – B client responsible for sending preprogramed messages to the server. B is responsible for sending the “add 2000” message first, followed by “transfer a 2000”

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class ClientB extends Client{


       // Constructor from superclass
    public ClientB() {
    }

    // Socket details
    static int ActionSocketNumber = 4545;
    static String ActionServerName = "localhost";
    static String ActionClientID = "Bravo";
    PrintWriter out;
    BufferedReader in;

       public static void main(String[] args) throws IOException,
InterruptedException {
             // Client initiation - run the object
             try {
                    // Initialise the BufferedReader and PrintWriter
              Socket ActionClientSocket = new Socket(ActionServerName,
ActionSocketNumber);
                    PrintWriter out = new
PrintWriter(ActionClientSocket.getOutputStream(), true);
                    BufferedReader in = new BufferedReader(new
InputStreamReader(ActionClientSocket.getInputStream()));

                 System.out.println("Initialised Client " + ActionClientID + "
and IO connections");
                 System.out.println("Welcome to Client "+ ActionClientID + "
console...");

                    String fromUser = "", fromServer = "";

                 // We can pre-programme some input here for our clients...
                 // Add 2000 Transaction
                 fromUser = "add 2000";
                 Thread.sleep(Random_Time());
              System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
              // bankServerThread.Crit_Section(fromUser, fromServer, out); // <--
HERE - IT ONLY WORKS IN RUN COS ITS NOT A STATIC MAIN - THIS STATIC IS KILLING ME
              out.println(fromUser);
              fromServer = in.readLine();
              System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

                 // Transfer 2000 Transaction
                 fromUser = "transfer a 2000";
                 Thread.sleep(Random_Time());
              System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
              out.println(fromUser);
              fromServer = in.readLine();
              System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

              ActionClientSocket.close();
              in.close();
              out.close();

        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: localhost ");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: "+
ActionSocketNumber);
            System.exit(1);
        }

    }

}
```

ClientC – C client responsible for sending preprogramed messages to the server. C is responsible for sending the “subtract 30.5” message first, followed by “transfer a 500”

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class ClientC extends Client{


      // Constructor from superclass
    public ClientC() {
    }

    // Socket details
    static int ActionSocketNumber = 4545;
    static String ActionServerName = "localhost";
    static String ActionClientID = "Charlie";
    PrintWriter out;
    BufferedReader in;

      public static void main(String[] args) throws IOException,
InterruptedException {
             // Client initiation - run the object
             try {
                    // Initialise the BufferedReader and PrintWriter
              Socket ActionClientSocket = new Socket(ActionServerName,
ActionSocketNumber);
                    PrintWriter out = new
PrintWriter(ActionClientSocket.getOutputStream(), true);
                    BufferedReader in = new BufferedReader(new
InputStreamReader(ActionClientSocket.getInputStream()));
```

```
                 System.out.println("Initialised Client " + ActionClientID + "
and IO connections");
                 System.out.println("Welcome to Client "+ ActionClientID + "
console...");

                   String fromUser = "", fromServer = "";

                 // We can pre-programme some input here for our clients...
                 // Subtract 30.5 Transaction
                 fromUser = "subtract 30.5";
                 Thread.sleep(Random_Time());
              System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
              out.println(fromUser);
              fromServer = in.readLine();
              System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

                 // Transfer 500 Transaction
                 fromUser = "transfer a 500";
                 Thread.sleep(Random_Time());
              System.out.println(ActionClientID + " sending message [" + fromUser
+ "] to Bank Server");
              out.println(fromUser);
              fromServer = in.readLine();
              System.out.println(ActionClientID + " received [" + fromServer + "]
from Bank Server");

              ActionClientSocket.close();
              in.close();
              out.close();

        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: localhost ");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: "+
ActionSocketNumber);
            System.exit(1);
        }

    }

}
```