CZ4013: Distributed Systems

Lab Report

Name	Matriculation ID	Contribution
Datta Anusha	U822948G	33.3%
Ravishankar Amrita	U1822377F	33.3%
Truong Cong Cuong	U1820080D	33.3%

Presentation time slot: April 5, Tuesday 9:00 - 9:15

School of Computer Science and Engineering (SCSE)

Table of Contents

1. Introduction

The objective of this project is to design and implement a distributed banking system based on the client-server architecture. In this project, we utilise our knowledge of interprocess communication and remote invocation to facilitate communication between clients and a server using the User Datagram Protocol (UDP). The client is able to perform multiple actions such as opening a bank account, closing an account, depositing/withdrawing money etc.

1.1 Environment Used

In this project, we utilise the following environment for successful communication between the client and the server. The following are necessary to run the system:

1. Programming Language - The server and the client were both developed using Java.
2. Operating System - The system was developed and tested on Windows machines.

1.2 Assumption

Several assumptions have been made while developing the system, they are as follows:

• User Interface - The entire user interaction in the system has been performed on the command-line interface, i.e. no graphical user interface has been made use of. The interface provides detailed responses and asks clear prompts to carry out the tasks successfully.

• Request Concurrency - We assume that the server handles user requests in a sequential manner i.e. only one request is performed at a time. Further, we assume that while the user is monitoring updates to the entire bank system, the user does not request any other services.

• Storage - In this system, no persistent storage exists, all the information is contained in the memory of that particular session.

• Non-negative balance - We assume a user's balance can never be negative. This is due to the reason that while performing error handling, functions return negative values to indicate erroneous inputs, and if everything is accurate they return the true balance.

• User Input - Minimal error checking is performed on the user input to the system. The system assumes that the input provided by the users are valid. For example, while accepting user input for the currency of their account, we assume that the user enters only "USD" or "SGD".

• Same currency transfer - While performing bank transfer of funds, we assume that the user transfers funds from their account in their original currency. That is, if an account was created with the initial currency as "SGD", we assume transfers from this account are made in SGD.

However, in case funds are transferred to an account with “USD” instead, we perform conversion from “SGD” to “USD” before depositing it into the other account.

2. System Design

2.1 Marshalling/Unmarshalling

All interprocess communication between client and server, irrespective of the chosen transport protocol, is transmitted as a sequence of bytes. Subsequently, the data involved in this communication must be subject to flattening (converted to a sequence of bytes) prior to transmission, and must be rebuilt upon arrival (retrieve information from sequence of bytes). To avoid issues of inconsistent data representations at client and server, such as Big Endian or Little Endian, External Data Representation (EDR) is utilised. EDR is a pre-agreed standard by both client and server for representation of data structures and primitive values.

This process of conversion of data to and from EDR for the purpose of transmission is formalised as Marshalling and Unmarshalling. The object to byte conversion follows the given specification:

Data Type	**Bytes**
Integer	4 bytes
Float	4 bytes
Double	8 bytes
String	4 + X bytes, where X is length of string (+ 4 required to store the length of the string)
OneByteInt	1 byte
ByteArray	4 + X bytes, where X is length of array (+ 4 required to store the length of the array)

2.2 System Services

This section outlines the design and features of the system services offered by our distributed banking system.

2.2.1 Service 1: Open New Account

This service allows a user to create a new account in the system, with the following specifications:

Request: Input Parameters	
Data	**Type**
User Name	String
Password	String
Currency (USD/SGD)	String
Initial Balance	Double

Upon processing of the request, the server replies with the following appropriate response:

Response: Return Result		
Condition	**Data**	**Type**
User account successfully created	User Account Number	Integer
User account not successfully created	`“ERROR: Account creation has failed.”`	String

2.2.2 Service 2: Close Existing Account

This service allows a user to close an existing account in the system, with the following specifications:

Request: Input Parameters	
Data	**Type**
User Name	String
User Account Number	Integer
Password	String

Upon processing of the request, the server replies with the following appropriate response:

Response: Return Result		
Condition	**Data**	**Type**

User account successfully closed	Success Message	String
User provided incorrect password	`"Error: Invalid Password. Please try again"`	String
User provided account number not registered under user name	`"Error: Invalid Account No. Please try again."`	String
User provided account number does not exist	`"Error: Invalid Account No. Please try again."`	String
User account not successfully closed	`"Error: Your request to close account has failed"`	String

2.2.3 Service 3: Deposit/Withdraw Funds

This service allows a user to deposit/withdraw funds into/from an existing account in the system, with the following specifications:

Request: Input Parameters	
Data	**Type**
User Name	String
User Account Number	Integer
Password	String
Currency (USD/SGD)	String
Deposit/Withdraw Amount	Double

For this service, the appropriate currency conversion is done to obtain the withdrawal/deposit amount in the same currency of the user account before the deposit/withdrawal. For instance, if the user account currency is SGD with initial balance 100SGD, a deposit of 50USD would be converted to a deposit of 67.84USD before the deposit operation is carried out. This shall result in an updated balance of 167.84SGD. Moreover, it may be noted that the conversion rates used in the program are static and not updated in real time. Upon processing of the request, the server replies with the following appropriate response:

Response: Return Result		
Condition	**Data**	**Type**
Deposit/Withdraw successful	Updated User Account Balance	Double
User provided incorrect password	`"Error: Invalid Password. Please try again."`	String

User provided account number does not exist	`"Error: Invalid Account No. Please try again."`	String
User provided deposit/withdrawal amount greater than current account balance	`"Error: You have insufficient funds. Your current balance is: X"`	String
Deposit/Withdraw not successful	`"Error: Withdraw/Deposit from/to your account has failed"`	String

2.2.5 Service 4: View Account Balance

This idempotent service allows a user to view account balance for an existing account in the system, with the following specifications:

Request: Input Parameters	
Data	**Type**
User Account Number	Integer
Password	String

Upon processing of the request, the server replies with the following appropriate response:

Response: Return Result		
Condition	**Data**	**Type**
Account Balance checked successfully	Account Balance	Double
User provided incorrect password	`"Error: Invalid Password. Please try again."`	String
User provided account number does not exist	`"Error: Invalid Account No. Please try again."`	String

2.2.6 Service 5: Transfer Funds

This non-idempotent service allows a user to transfer funds from one existing account to another in the system, with the following specifications:

Request: Input Parameters	
Data	**Type**
User Name	String
User Account Number	Integer

Password	String
Recipient Account Number	Integer
Transfer Amount	Double

Note that the assumption here is that the sender shall specify the transfer amount in the currency of their own account, and any necessary conversion shall be conducted for the transferred amount on the recipient's end. For instance, consider the user account currency is SGD with initial balance 100SGD, and the recipient account currency is USD with an initial balance of 100USD. In this case, the transfer amount specified would be considered in SGD, which is the sender's currency. Hence, a transfer of 20SGD shall be converted to 14.74USD before the transfer operation is carried out. This would result in an updated balance of 80SGD for the sender and 114.74SGD for the recipient. Similar as noted above, the conversion rates used in the program are static and not updated in real time. Upon processing of the request, the server replies with the following appropriate response:

Response: Return Result		
Condition	**Data**	**Type**
Transfer successful	Updated User Account Balance	Double
User provided incorrect password	`"Error: Invalid Password, Please try again."`	String
User provided account number does not exist	`"Error: Invalid Account No. Please try again."`	String
User provided transfer amount greater than current account balance	`"Error: You have insufficient funds. Your current balance is: X"`	String
Transfer unsuccessful	`"Error: The requested fund transfer has failed"`	String

2.2.4 Service 6: Monitor Updates

This service allows a user to monitor all updates across the banking system for a specified interval of time, with the following specifications:

Request: Input Parameters	
Data	**Type**
Monitor Interval Length (in minutes)	Integer

This functionality is achieved via a callback. Upon processing of the request, the server registers the client and sends an acknowledgment response `"Auto-monitoring registered. Waiting` `for updates..."` Subsequently, the server sends a callback to the client for any system updates that occur within the specified monitor interval:

Response: Return Result		
Service Name	**Data**	**Type**
Open New Account	Same as outlined in service responses above	
Close Existing Account		
Deposit/Withdraw Funds		
View Account Balance		
Transfer Funds		
Socket Exception	`"ERROR: Socket timeout exception in callbackUpdates handler"`	String

Furthermore, the server also provides a notification to the client upon expiration of the monitoring interval stating `"Auto-monitoring expired. Please subscribe again if you wish to extend the period".` There is message logging implemented on the server side as well to facilitate the traceability of the program, with messages such as `"New subscriber added!"`, `"Client already exists in list of subscribers."` or `"Removing subscriber."`

2.3. Fault Tolerance

The two main communication protocols that exist are: TCP and UDP. UDP is suitable for time-sensitive applications and for high-speed query-response protocols. It has been designed for low latency and loss-tolerating connections. To achieve the above, UDP does not have a handshaking procedure like TCP. Consequently, it is unable to provide reliability. Thereby, the application layer must perform fault tolerance to provide reliability to the system.

2.3.1 Invocation Semantics

Failures can be handled primarily in three ways: request message retransmission, duplicate filtering, and method re-execution/reply retransmission. In this project, we observe two invocation semantics, i.e. At-Least-Once and At-Most-Once.

2.3.1.1 At-Least-Once

At-Least-Once invocation semantic utilises request retransmission, but not duplicate filtering. In this semantic, duplicate messages cannot be distinguished as duplicate. Such duplicate requests are handled by re-executing the method. This re-execution can cause failure for non-idempotent operations. In our system, we perform retransmission whenever a response is not obtained within a specific duration of time.

2.3.1.2 At-Most-Once

At-Most-Once invocation semantic utilises request retransmission and duplicate filtering. In this semantic, if a duplicate message is received, the exact response as the previous time is re-sent. In our system, we utilise messageID to differentiate between messages and to check if it is a duplicate. The server utilises a combination of client address and messageID to distinguish between messages. The first time a specific unique request is received (as distinguished by the messageID or the client address and messageID in case of server), its intended reply is saved.

2.3.2 Comparison

We compare the fault tolerance of the two invocation semantics for idempotent and non-idempotent operations. Idempotent operations refer to those operations that can be performed repeatedly with the same effect as if performed exactly once.

2.3.2.1 Idempotent

We utilise Service 5, “View Account Balance”, to observe the fault tolerance of the two invocation semantics when handling idempotent operations. Both the invocation semantics are able to successfully handle the failures in case of idempotent operations. This is further explained in Section 3.

2.3.2.2 Non-Idempotent

We utilise Service 6, “Transfer Funds” to observe the fault tolerance of the two invocation semantics when handling **non-idempotent** operations. The At-Least-Once invocation semantic produces an arbitrary failure, i.e. the transfer service is executed more than once thereby causing unwanted results. In contrast, the At-Most-Once invocation semantic is able to handle the failure successfully. This is further explained in Section 3.

3. Experiments:

3.1. Invocation semantics

We experiment with 2 invocation semantics on the server side: at-least-once and at-most-once.

For each invocation semantic, we tested it with the non-idempotent operation (Service 6: Transfer of money) and idempotent operation (Service 5: Checking of user balance) under different scenarios: request loss and reply loss. The description for the scenarios are as follows:

- Request loss scenario: We created a client and a server. The **client** would be tested with different probabilities of message **sending failures**: 30%, 50% and 70%. The server has a normal socket which always sends and receives messages successfully. The client has a socket timeout of 5 seconds.

- Reply loss scenario: We create a client and a server. The client has a normally functioning socket, it always sends and receives messages successfully. The client has a socket timeout of 5 seconds. In contrast, the **server** has a SendingLossSocket with a probability of **failure in sending** messages to the client of 70%.

3.1.1. At-least-once invocation semantics

3.1.1.1 Non-idempotent operation (Transfer of Funds between accounts)

3.1.1.1.1 Request loss:

In this scenario, we experimented with a situation where one account requested to transfer a certain amount of money to another account. Due to the sending failure on the client side, the client had to send the request multiple times to the server. However, because the timeout was 5 seconds (much larger than the time required for processing of the client request and sending the appropriate reply by the server), once the first request successfully reached the server, the reply was received by the client immediately and the client did not send any more requests. With that, only one “transfer funds” operation was executed and the account balances were updated correctly.

1.1.2. Reply loss:

Two accounts were created: one for Bob, with an initial balance of 150 SGD, another for Alice, with an initial balance of 1000 SGD. Alice requested the server to transfer 70 SGD to Bob. However, due to the sending loss on the server side, the server’s replies were lost thrice, thus causing the client to resend the request thrice. Whenever the server received the request from the client, it executed the request **again**. After the fourth request, a reply was sent successfully to the client. After the fourth request, Alice's final balance was: 1000 - 70*4 = 720 SGD and Bob's final balance was: 150 + 70 * 4 = 430 SGD. The expected final balance was 930 SGD for Alice and 220 SGD for Bob. This highlights the issues that the at-least-once invocation semantic is unable to handle.

1.2. Idempotent operation: Checking Balance

Since idempotent operations can be performed repeatedly with the same effect as performing it once, in both the scenarios of reply loss and request loss, the system functioned as expected, deducting and adding the correct amounts from/to the accounts and showing the correct final balance for each user.

3.1.2. At-most-once invocation semantic

3.1.2.1. Non-idempotent operation: Transfer

3.1.2.1.1.Request loss

Similar to the case for 3.1.1.1., because the request execution time and time for the reply to reach the client by the server is much shorter than the client socket timeout, finally once a request reached the server successfully, the client received the reply immediately and did not send any request any more. One transfer was executed and accounts were updated correctly.

3.1.2.1.2. Reply loss

Two accounts were created: one for Bob, with an initial balance of 150 SGD, another for Alice, with an initial balance of 1000 SGD. Alice requested the server to transfer 70 SGD to Bob. In contrast to the at-least-once invocation semantic, even though the server's replies were lost thrice, the server stored the reply for the specific request's message ID. The server just re-transmitted the reply to the client upon receiving the same request again. Therefore, when the client sent the request to the server for the fourth time, the reply was successfully sent to the client and the balances of Bob and Alice were found to be updated as expected, i.e., Alice's final balance was 930 SGD and Bob's final balance was 220 SGD.

3.1.2.2. Idempotent operation: View Account Balance

Since idempotent operations can be performed repeatedly with the same effect as performing it once, in both the scenarios of reply loss and request loss, the system functioned as expected. Throughout our experiments with both the reply and request loss scenarios, viewing account balance was executed successfully under at-most-once invocation semantics.

3.2. Error Handling

We tested the following services and their user input error handling:

Service	**Error handling implemented and tested**
Closing of accounts	● Account number does not exist ● Account number does not matches the account holder name ● Invalid password
Balance Transfer	● Invalid account number ● Invalid password ● Insufficient funds ● Account number does not match the account holder name
Balance Update (Deposit/ Withdrawal)	● Invalid account number ● Invalid password ● Insufficient funds (in case of withdrawal) ● Account number does not match the name
Check Balance	● Invalid account number ● Invalid password

4. Conclusion

In this project, we have designed and implemented a distributed banking system based on client-server architecture with UDP as the transport protocol. This system offers multiple features including opening an account, closing an existing account, depositing or withdrawing money and monitoring updates via callbacks. Additionally, it offers an idempotent feature of checking balance and a non-idempotent feature of performing a bank transfer between two accounts. Lastly, we have also conducted experimentation to determine the robustness of our inbuilt error handling and the effects of different invocation semantics on fault-tolerance.