NANYANG TECHNOLOGICAL UNIVERSITY

CZ4013

DISTRIBUTED SYSTEMS

Assignment Report

Author:	*Matric. no.:*
Eric Leonardo Lim	U1420158D
Gede Bagus Bayu Pentium	U1420090G

April 4, 2018

Contents

1 Introduction 2
1.1 Environment . . . 2
1.2 Assumption . . . 2

2 Design 3
2.1 Architecture Design . . . 3
2.1.1 Server . . . 3
2.1.2 Client . . . 4
2.2 Communication Design . . . 4
2.2.1 Message Structure . . . 4
2.2.2 Marshalling/Unmarshalling Primitive Datatypes . . . 5

3 Services 6
3.1 Service 1: Open New Account . . . 6
3.2 Service 2: Close Existing Account . . . 7
3.3 Service 3: Deposit/Withdraw Money . . . 7
3.4 Service 4: Monitor Update . . . 7
3.5 Service 5: Change Password . . . 8
3.6 Service 6: Transfer Money . . . 8

4 Fault Tolerance 9
4.1 Invocation Semantics . . . 9
4.1.1 At-Least-Once . . . 9
4.1.2 At-Most-Once . . . 9
4.2 Comparison . . . 10
4.2.1 Idempotent . . . 10
4.2.2 Non-Idempotent . . . 10

5 Conclusion 11

1 Introduction

This project consolidates basic knowledge about interprocess communication and remote invocation through constructing client and server programs that use UDP as the transport protocol. The client and server programs are designed to simulate a distributed banking system.

1.1 Environment

The following are the necessary environment to run the system.

- **Operating System** — The banking system is developed and tested in a GNU/Linux environment.
- **Programming Language** — There are two languages used to develop the system. The server is developed using `C++` and the client is developed using `Java`.

1.2 Assumption

Several assumptions have been made in developing the system, as follows:

- **User Interface** — The user interaction to the system is fully done in command-line interface (no Graphical User Interface).
- **User Input** — There are minimum error checking in user input to the system. The system assume all the user input are valid.
- **Request Concurrency** — Server and client are handling the user input in sequential order. The system assume only one request at a time is performed.
- **Client-Server Communication** — All clients are assumed to know server address and port number.
- **Storage** — This system does not have any persistent storage (All information are contained in memory).

2 Design

2.1 Architecture Design

The system used UDP as communication protocol between client and server.

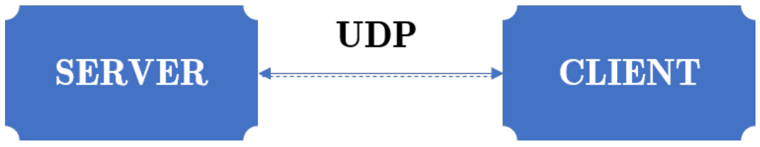

Figure 1: Communication between client and server

2.1.1 Server

The server architecture can be seen in Figure 2.

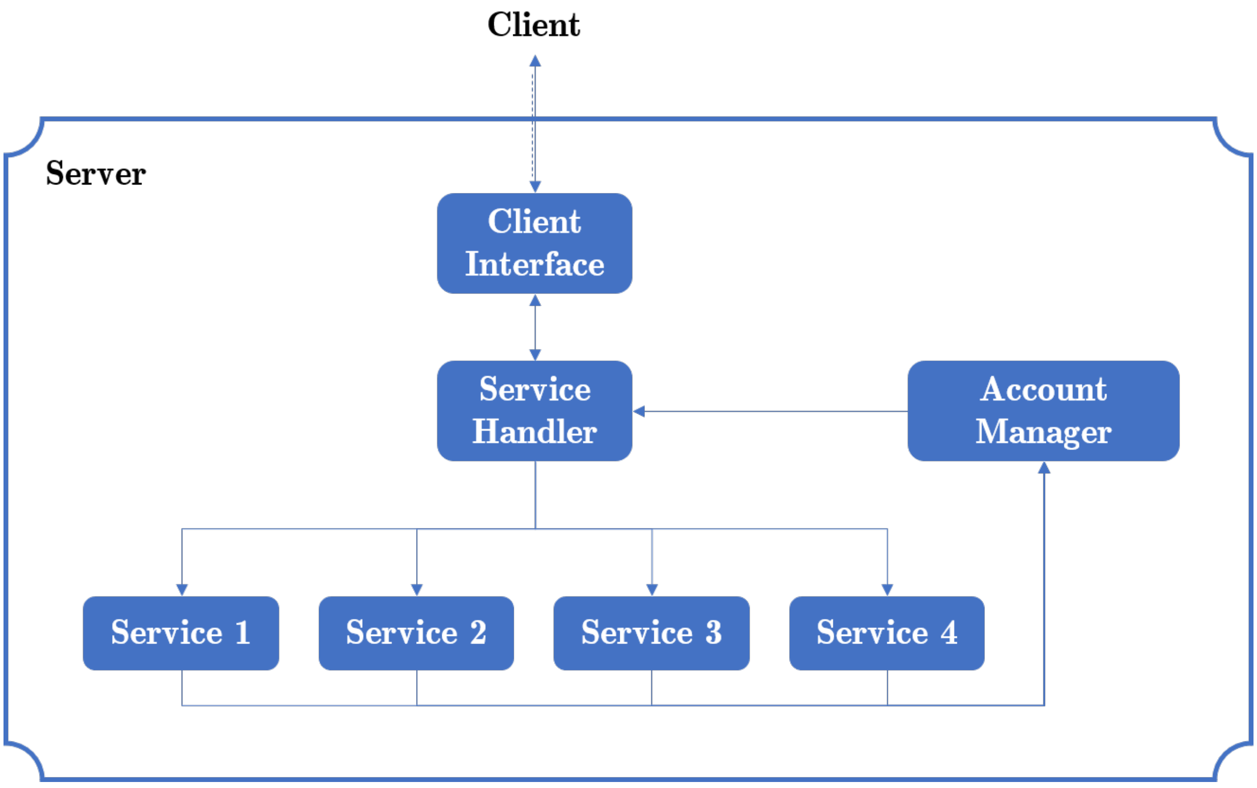

Figure 2: Server architecture

2.1.2 Client

The client architecture can be seen in Figure 3

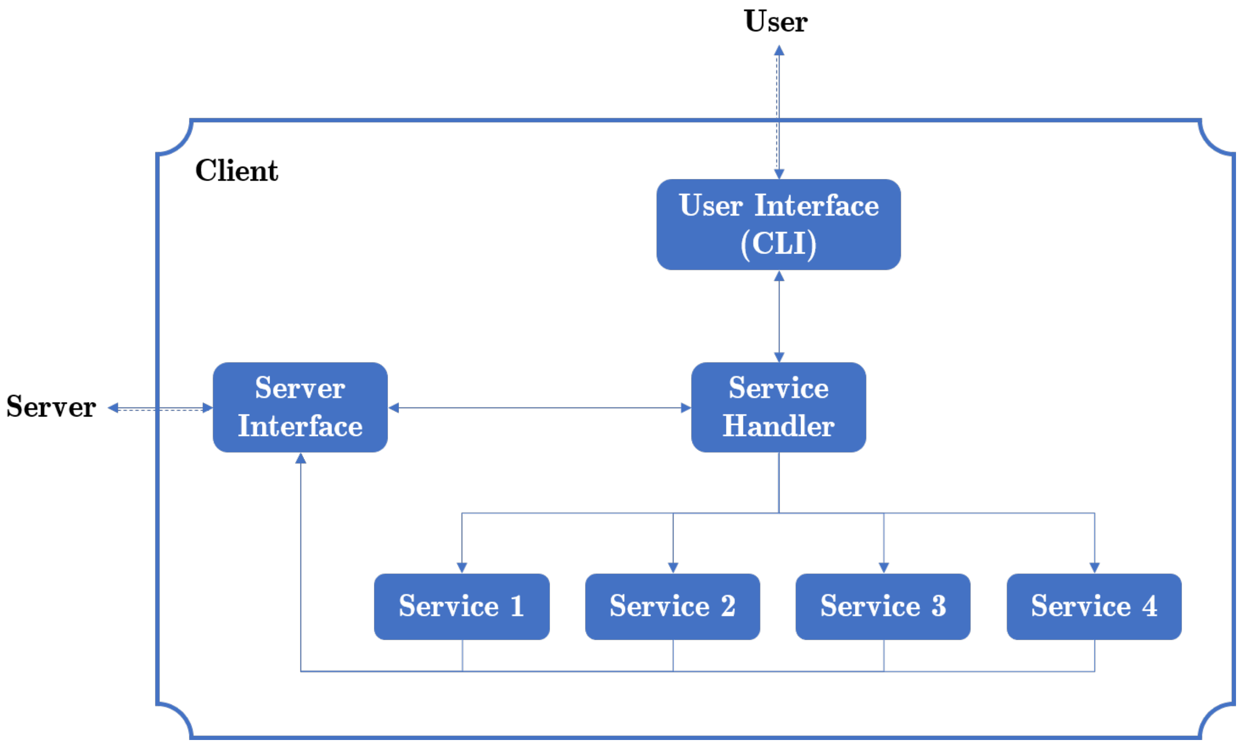

Figure 3: Client architecture

2.2 Communication Design

2.2.1 Message Structure

Each message (request/response/acknowledgement) is performed with two transmissions, i.e. transmission of header followed by transmission of content.

The header size is constant, i.e. 4 bytes. It is an integer which represents the size of the actual content. This allows the recipient to adjust the buffer size for receiving the content.

The content message format varies on whether it is a request, response, or an acknowledgement. Generally, for request messages, the format is divided into sections, such as request ID and data, consecutively. For response messages, the format is divided into response ID, status, and data. For acknowledgement, it is divided into response ID and status.

The data section format depends on the service type. Each service type requires passing a message consisting of a few primitive data-types, such as integers, floating-point values, and strings. Each data to be passed is converted into byte-arrays and is prepended with the number of bytes required, and then concatenated into one, i.e. the data section of the message.

The structure for each service will be described in detail in Section 3.

2.2.2 Marshalling/Unmarshalling Primitive Datatypes

Integer values Since `int` is used in both server (C++) and client (Java) programs, integer values are represented in 4 bytes and is able to represent negative values.

Marshalling an integer into a byte array of size 4 is as simple as obtaining each octet and then assigning the octet to the corresponding index in the byte array. Since in both C++ and Java, when assigning an `int` into a `byte`, only the last 8 bits of the integer will be captured, each octet in the integer can be obtained by right-shift operations. To obtain the i^{th} octet, the integer is right-shifted $8 \cdot (4 - i)$ number of times.

Unmarshalling the byte array into an integer value can be achieved by performing left-shift operations on the values of each index i of the byte array, with each value of index i to be left-shifted $8 \cdot (4 - i)$ number of times, and then performing `OR` operations to combine them into one integer.

Floating-point values We are using `float` to represent all floating-point values in both sever and client programs, so each floating-point value is 32-bit. Since `float` in C++ is implemented in the same way as it is in Java , i.e. as 32-bit precision IEEE 754 floating points, we can perform marshalling and unmarshalling on floating-point values in the same way as doing so on integer values.

Enumerated type values We represent enumerated type values as integer values, so marshalling and unmarshalling is performed in exactly the same way.

Strings By employing the aforementioned message structure, both variable-length and fixed-length strings can be represented uniformly, i.e. only as byte-arrays with each byte representing each character.

3 Services

Request The general format of all the request is:

- *ID* — [`4 bytes`] request id generated by the client.
- *Service type* — [`4 bytes`] type of request send from the client.
- *Header of Content 1* — [`4 bytes`] header length of content 1.
- *Content 1* — [`Variable length`] content 1 of the request.
- *Header of Content 2* — [`4 bytes`] header length of content 2.
- *Content 2* — [`Variable length`] content 2 of the request.
- ...

Response The general format of all the response is:

- *ID* — [`4 bytes`] response id generated by the server.
- *Status* — [`1 byte`] response status
- *Header of Content 1* — [`4 bytes`] header length of content 1.
- *Content 1* — [`Variable length`] content 1 of the response.
- *Header of Content 2* — [`4 bytes`] header length of content 2.
- *Content 2* — [`Variable length`] content 2 of the response.
- ...

3.1 Service 1: Open New Account

This service allows users to create a new account in the system by specifying *name*, *password*, *default currency*, and *initial balance*. If the request is successfully processed, the server replies with a response with the *account number* assigned to the account.

The request message of this service has the following format:

Content	**Parameter**	**Type**
1	Name	Variable-length string
2	Password	Fixed-length String
3	Currency	Enumerate
4	Balance	Float

The response of this service has the following format:

Content	**Parameter**	**Type**
1	Account Number	Integer

3.2 Service 2: Close Existing Account

This service allows user to close an existing account in the system by specifying *name*, *account number*, and *password*. If the request is successfully processed, the server response with acknowledgment.

The request of this service has the following format:

Content	Parameter	Type
1	Name	Variable-length string
2	Account Number	Integer
3	Password	Fixed-length string

The response of this service is only acknowledgment whether the service is successful or not. This acknowledgement is represented by the response status.

3.3 Service 3: Deposit/Withdraw Money

This service allows user to deposit or withdraw money to their account by specifying *name*, *account number*, *password*, *currency* and *balance*. If the request is successfully processed. the server response with acknowledgment and remaining balance in the account.

The request of this service has the following format:

Content	Parameter	Type
1	Name	Variable-length string
2	Account Number	Integer
3	Password	Fixed-length string
4	Currency	Enumerate
5	Balance	Float

The response of this service has the following format:

Content	Parameter	Type
1	Currency	Enumerate
2	Updated Balance	Float

3.4 Service 4: Monitor Update

This service allows the user to monitor update in the system by specifying the *duration* of the monitor in milliseconds. If the request is successfully processed, the server response with acknowledgment and subsequently sending a callback to the client for each update happens during the *duration* specified.

The request of this service has the following format:

Content	Parameter	Type
1	Duration	integer

The response of this service has the following format:

Content	Parameter	Type
1	Update	String

3.5 Service 5: Change Password

This service allows user to change password by specifying *name*, *account number*, *old password*, and *new password*. If the request is successfully processed, the server response with acknowledgment.

The request of this service has the following format:

Content	Parameter	Type
1	Name	Variable-length string
2	Account Number	Integer
3	Old Password	Fixed-length string
4	New Password	Fixed-length string

The response of this service is only acknowledgment whether the service is successful or not. This acknowledgement is represented by the response status.

3.6 Service 6: Transfer Money

This service allows user to transfer money to another user by specifying *name*, *account number*, *recipient name*, *recipient account number,password*, *currency*, and *balance*. If the request is successfully processed, the server response with acknowledgment and remaining balance in the account.

The request of this service has the following format:

Content	Parameter	Type
1	Name	Variable-length string
2	Account Number	Integer
3	Recipient Name	Variable-length string
4	Recipient Account Number	Integer
5	Password	Fixed-length string
6	Currency	Enumerate
7	Balance	Float

The response of this service has the following format:

Content	Parameter	Type
1	Currency	Enumerate
2	Updated Balance	Float

4 Fault Tolerance

The UDP is an alternative communication protocol to TCP. It is mainly designed for low latency and loss tolerating connections. To achieve low latency, UDP has no handshaking dialogues, thus exposes the protocol to any unreliability. Hence, the application layer needs to perform fault tolerance to achieve reliable communication.

4.1 Invocation Semantics

There are three techniques to handle failures, such as request message retransmission, duplicate filtering, and method re-execution/reply retransmission. In this project, we observe two out of them, i.e. At-Least-Once and At-Most-Once invocation semantics.

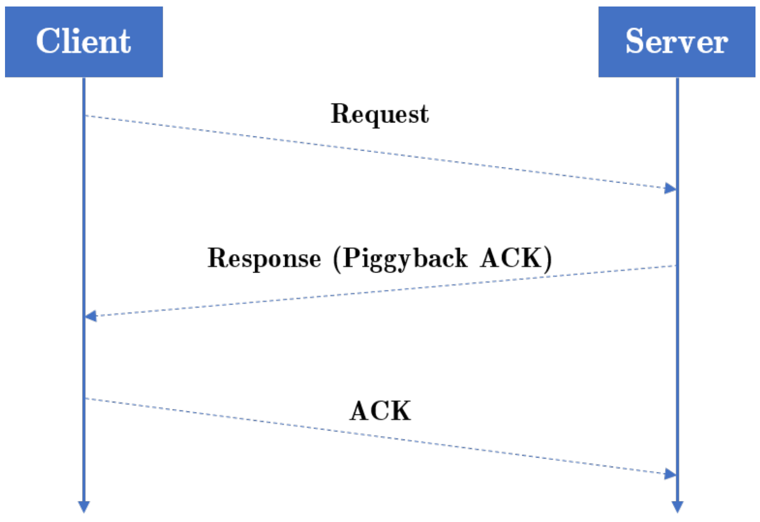

Figure 4: Client-server handshake

4.1.1 At-Least-Once

At-Least-Once semantic features request retransmission, but no duplicate filtering. Upon receiving duplicate messages, it cannot distinguish them as duplicates, so duplicate requests are handled by re-executing method. In our implementation, retransmission is performed whenever no reply is obtained within a particular period of time.

4.1.2 At-Most-Once

At-Most-Once semantic features request retransmission and duplicate filtering. Upon receiving duplicate messages, it resends the exact same reply that has already been sent before. In our implementation, message ID is used to distinguish whether two messages are duplicates.

In case of server, the client address is also used in addition to the message ID. Upon the first time receiving a particular unique message as distinguished by the ID (and the client address, in case of server), the intended reply is saved.

4.2 Comparison

4.2.1 Idempotent

We use Service 5 to observe the fault tolerance of the two invocation semantics when handling idempotent operations. As illustrated in table 1, the behaviors of the two invocation semantics when the response message failed to be sent to the client are alike, i.e. both are able to successfully handle the failures.

Table 1: Idempotent service fault tolerance result

Fault	**Invocation Semantic**	
	At-Least-Once	**At-Most-Once**
Request failure	client resends request	client resends request
Response failure	password is successfully changed	password is successfully changed
Acknowledgement failure	server resends response client resends acknowledgment	server resends response client resends acknowledgment

4.2.2 Non-Idempotent

We use our service 6 to observe the fault tolerance of the two invocation semantics when handling non-idempotent operations. As illustrated in table 2, the At-Least-Once invocation semantic produces an arbitrary failure, i.e. the transfer service is executed more than once. In contrast, the At-Most-Once invocation semantic is able to handle the failure successfully.

Table 2: Non-idempotent service fault tolerance result

Fault	**Invocation Semantic**	
	At-Least-Once	**At-Most-Once**
Request failure	client resends request	client resends request
Response failure	transfer is executed more than once	transfer is executed once
Acknowledgement failure	server resends response client resends acknowledgment	server resends response client resends acknowledgment

5 Conclusion

In this project, we have designed and implemented a distributed banking system using the client-server architecture that uses the UDP protocol to communicate between each other. We also experimented with how different invocation semantics affect fault-tolerance.