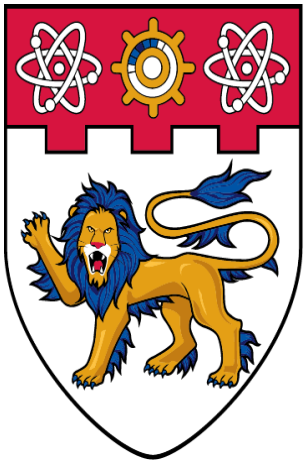

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE/CZ4013
Distributed System Project Report

Demo Slot Number: 26
Demo Time: April 4 2023, 14:15-14:30

Submitted By:

Name	Matriculation Number
Au Yi Xian	U1923991D
Cheng Gin Yee Shaun	U2020487L
Gareth Thong Jun Hong	U2021083G

Percentage of work done:

Name	Percentage (%)
Au Yi Xian	33.3
Cheng Gin Yee Shaun	33.3
Gareth Thong Jun Hong	33.3

Objective

The goal of this project is to design and implement a distributed flight information system based on client-server architecture and to establish knowledge about interprocess communication and remote invocation.

Client-Server Architecture Design

Network Diagram

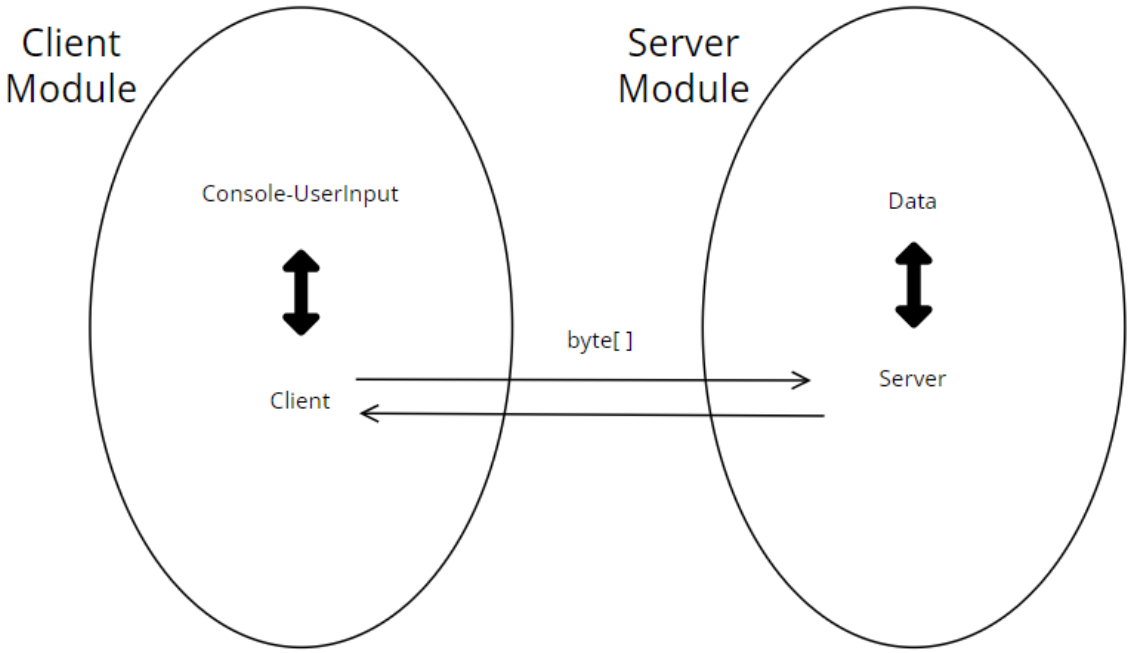

Figure 1. Network Implementation

Marshalling/ Unmarshalling Implementation

Rationale

The aim is to make the marshalling and unmarshalling function as general as possible, such that the same marshalling/unmarshalling functions can be reused in the client as well as the server for all messages.

To do this, an **AttributeValue Class** is instantiated to store a pair of attribute and value fields. To account for the different data types, different classes such as **AttributeValueString** (String), **AttributeValueInt** (Integer), **AttributeValueFloat** (Float), **AttributeValueDouble** (Double) and **AttributeValueBoolean** (Boolean) are extended from the AttributeValue parent class.
The Attribute and Value fields will be ordered together with other flags indicating various important information (explained below), like Attribute Length, to construct a **Attribute-Value pair structure**. The Attribute-Value pair structure will thus consist the following:

Attribute Length	Attribute	Value Type	Value Length	Value

Table 1. Attribute-Value Pair Structure

Attribute Length: This is the length of the attribute. Data type is Integer and 4 bytes are allocated.

Attribute: This is a string containing the attribute field corresponding to the value, such as “Departure Time”, “Airfare” or “Error Message”. Data type is String. A variable number of bytes is allocated depending on the string length.

Value Type: This is a string which indicates the data type of the value. “S” indicates a string, “I” indicates an integer and “F” indicates a float. “D” indicates a double. “B” indicates a Boolean. 1 byte is allocated as the string is always of one in length.

Value Length: This is the length of the value. Data type is Integer and 4 bytes are allocated.

Value: This is the value field. Number of bytes to be allocated is the length of the string if Value Type is “S”, 4 if the Value Type is “I” or “F”, 8 if the Value Type is “D” and 1 if the Value Type is “B”.

Marshalling Steps

1. Allocate a main buffer of 1024 bytes which will contain the marshalled bytes of the message.
2. Allocate a temporary buffer with the exact number of bytes of the Attribute-Value pair structure.
3. Convert the value of each Attribute-Value pair field in the Attribute-Value pair structure into bytes.
4. Put the bytes into the temporary buffer.
5. Copy the temporary buffer into the main buffer.
6. Repeat Steps 2 to 5 for every Attribute-Value pair.

The main buffer will contain all the Attribute-Value pairs in bytes.

Unmarshalling Steps

1. Start reading from the start of the byte buffer.
2. Read the next 4 bytes to get the value of the Attribute Length field.
3. Read the amount of bytes specified by the Attribute Length field to get the value of the Attribute field.
4. Read the next byte to get the value of the Value Type field.
5. Read the next 4 bytes to get the value of the Value Length field.
6. Read the amount of bytes specified by the Value Length field to get the value of the Value field.
7. Repeat steps 2 to 7 for every Attribute-Value pair.

Message Structure

Each message sent by either client or server is a list of Attribute-Value pairs.

Messages sent by the client will always include these two Attribute-Value pairs at the start of the message:

Attribute	Value

option	The menu option number of the service requested by the client.
	`Distributed Flight Booking System`
	`----------`
	`Select an option from [1-7]:`
	`1. Check Flight ID based on Source and Destination`
	`2. Check Departure Time, Airfare, Seat Availability via Flight ID`
	`3. Book Seats with Flight ID and Number of Seats`
	`4. Monitor Seat Availability`
	`5. Check Cheapest Flights via Source`
	`6. Increase/Decrease Airfare via Flight ID`
	`7. Set Invocation (1 for At Least Once, 2 for At Most Once)`
	`8. Exit`
	`----------`

requestId	The requestId is unique and is formed by the concatenation of the Request count + IP address + Port number

Messages sent by the server will always include at the start of the message:

Attribute	Value
Success/Error Flag	Informs the client whether the request has succeeded or failed

The remaining contents of the Attribute-Value pairs in the messages are specific to the service requested.

Client Implementation

The client provides a console-based interface which users can interact with to use the functionalities of our distributed flight booking system. It consists of 5 main classes, Main, ByteConverter, Marshaller, Functions and Utils.

The Main class contains the main menu of the interface which specifies the different options which the user can choose from. It also instantiates a socket to send requests and receive replies from the server. The server IP and port address which the client will interact with are also specified accordingly.

The ByteConverter class contains functions which convert Attribute-Value pairs into bytes and returns an array of bytes. This byte conversion process also accounts for the conversion of different data types.

The Marshaller class contains the marshalling and unmarshalling logic mentioned above. It uses the ByteConverter class to convert the fields into bytes.

The Functions class contains the processing logic for unmarshalled data according to the various functionalities of the flight booking system.

The Utils class contains reusable functions such as printing out data received from the server, sending data packets to the server and receiving data packets from the server.

Server Implementation

The server handles client requests and sends back replies to the client accordingly. The server also contains the data of the flights. It consists of 7 main classes: Main, Flight, Marshaller, ByteConverter, Functions, MonitorInfo, Utils. The Marshaller and ByteConverter classes are the same as the ones described in the client implementation.

The Main class initialises the flight data and opens a socket to receive requests from clients and send replies to clients. It stores the message history of client requests, which will be explained further in the Invocation Semantics section. It also stores the list of clients which are monitoring the seat availability of the flights.

The Flight class consists of the flight attributes, such as the flight identifier, departure time, source, destination, airfare and the number of seats available.

The Functions class contains the processing logic for unmarshalled data according to the various functionalities of the flight booking system.

The MonitorInfo class stores the information of the clients which are monitoring the seat availability of the flights. The information includes the time which the client starts to monitor the server, the monitor interval and the flight identifier of the flight being monitored. If the seat availability of the monitored flight is changed, the server will send the respective client/s information of the current number of seats available via a callback.

The Utils class is almost identical to the one described in the client implementation. The only exception is that it contains extra logic to simulate server packet loss, which is used in our invocation semantics experiments to demonstrate correct or wrong results.

Additional Operations

We implemented an additional idempotent and non-idempotent operation:

Idempotent Operation
This operation allows a user to obtain all flight destinations by specifying the source location

Non-Idempotent Operation
This operation allows a user to increase/decrease the airfare of a flight by specifying the flight identifier and the amount to increase/decrease

Invocation Semantics

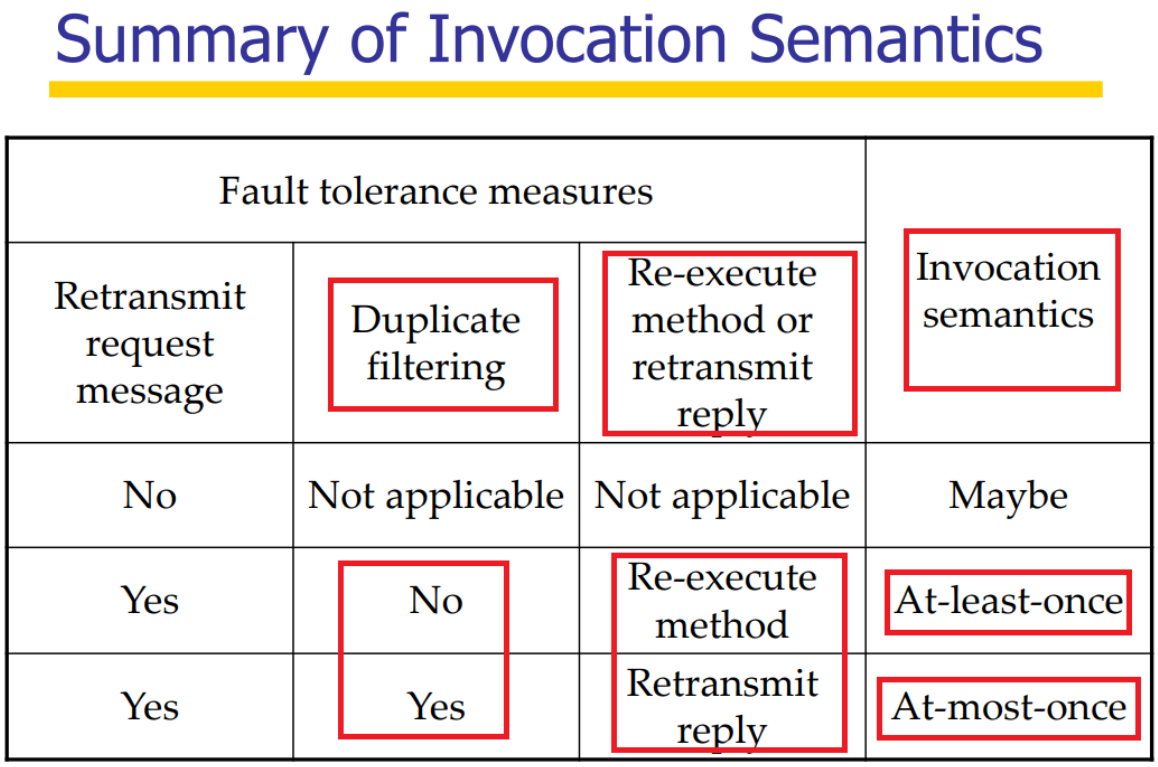

Figure 2. Difference between the 2 Invocation Semantics

By **default**, the server uses at-least-once invocation semantics upon start-up. However, the client can have the choice of switching between the

2 semantics (i.e. **at-least once or at-most once**) by choosing option 7 in the menu.

At-least once

As explained in Figure 2, in at-least-once invocation semantics, there is no need for duplicate filtering and thus no need to reference the message history of client requests stored in the server. To implement at-least-once semantics in our application, the client implements a timeout to resend request messages if no reply is received from the server. The server will always execute the operation upon receiving a request before sending a reply to the client.

In our experiments, there are two non-idempotent functions. The first non-idempotent function is the booking of seats by specifying the flight identifier and the number of seats to reserve. The second non-idempotent function, which we mentioned above, allows the client to alter the airfare by specifying the flight identifier and an amount to increase/decrease the airfare by. Since there is no duplicate filtering for this invocation semantic, our experiments performed on these two functions are able to yield wrong results.

Our implementation allows us to specify a user-defined server packet loss rate for these experiments with a default of **50%**. Upon server packet loss, the client will repeatedly send requests to the server until it receives a reply. This may cause the functions to be executed more than once on the server. In the first non-idempotent function, the server will reserve more seats than the quantity requested by the client. Whereas for the second non-idempotent function, the airfare of the flight will be decreased or increased more than the amount specified by the client.

At-most once

At-most-once invocation semantics requires us to reference the message history of client requests stored in the server to filter duplicate requests, in order to prevent the re-execution of any operation, as shown in Figure 2. To implement at-most-once invocation semantics, we store the requestId of each client request in the message history. With this unique identifier, the server can identify duplicate requests (i.e. same requestId) from a client. If a duplicate request is received, the server will not re-execute the operation and will simply resend the reply message to the client. This is very important for non-idempotent functions as they will result in incorrect results if executed more then once. Note that the timeout implemented in the client always causes the client request to be resent if no reply is received from the server, regardless of the invocation semantics.

For the first non-idempotent function, the server will always reserve only the number of seats requested by the client. For the second non-idempotent function, the server will increase/decrease the airfare by only the amount requested by the client. The experiments on non-idempotent functions we described previously will always yield correct results even if the server receives duplicate requests.