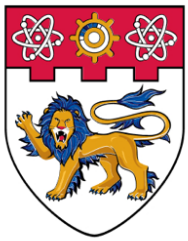

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

School of Computer Science and Engineering

Nanyang Technological University, Singapore

CE/CZ4013: Distributed Systems Course Project

Distributed Flight Information System

Name	Matriculation Number	Contribution
Acharya Atul	U1923502C	33.33%
Goel Tejas	U1923301G	33.33%
Arora Srishti	U1922129L	33.33%

March 30, 2023

Table of Contents

1. Introduction

The objective of the project is to design and implement a distributed flight information system based on client-server architecture. The system allows multiple clients to request services from the Flight Information server. The **UDP Protocol** is followed for inter-process communication. **We used C++ to code the client and Java to code to server**.

2. System Design

The Flight Information System consists of two components – the client which requests flight services, and the server which provides flight services. The server and client communicate with each other using the UDP protocol.

Both the server and the client use the **Factory design pattern** to represent the entities. Each entity is an interface with a corresponding servant class that implements the entity interface. The server additionally declares and implements factory interfaces for each entity which return and maintains state of all entity objects. This pattern ensures appropriate abstraction and loose coupling in the system design.

2.1. Client Architecture

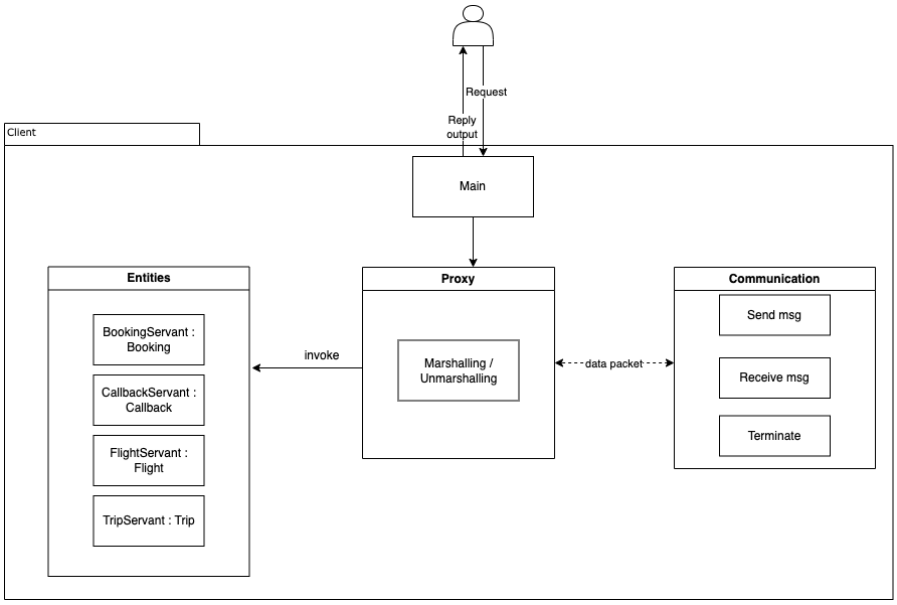

Figure 1. Flight System Client Architecture

The user selects a service from the main menu and provides inputs for the required fields. These inputs are passed to the proxy of the respective service where they are marshalled. The proxy then calls the *send_msg* method from the Communication package to transmit the input data packet to the server. Further, the proxy calls the *receive_msg* method and waits for the server to send a reply. Once a reply is received, the proxy unmarshalls it and passes it to the respective entity to display the output. If no reply is received within a threshold of 3 seconds, the proxy retransmits the request to the server.

2.2. Server Architecture

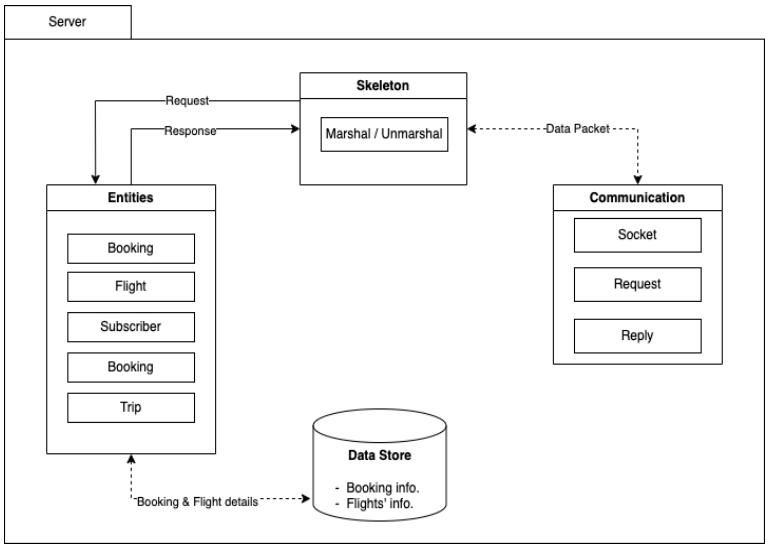

Figure 2. Flight System Server Architecture

Multiple clients can connect with the server through a socket connection and make requests. Each request is handled sequentially by the main program, which then routes the request to the correct service skeleton. The main program unmarshalls the service id received in the data packet from a client and passes the data contents to the skeleton of the respective services. The skeleton unmarshalls the data contents and passes the inputs as parameters to the suitable methods from the respective entity's servant class. The entities read and write to the data store for querying and updating information about bookings and flights. The output is returned to the skeleton where it is marshalled and communicated back to the client.

3. Marshalling and Unmarshalling

The **proxy class of the client and the skeleton class of the server** are responsible for marshalling and unmarshalling all messages received from the om communication module.

Marshalling and Unmarshalling of data are done by first determining the endianness of the machine. We implemented functions in the client and the server to check for that. We assume that the network always uses Big-Endian byte ordering. We learnt that all Java programs use Big-Endian ordering for its data types and hence the *isLittleEndian* function that we use in the server program (Java) always returns false. In the client program, if the host machine uses Little Endian ordering, we reverse the byte ordering before sending and after receiving a message over the network. To check for endianness in the client (C++), we coded a function to check the MSB of the unsigned integer 1 when cast into a character.

The general approach used for marshalling and unmarshalling in this project is as follows:

1. We have defined functions for processing primitive data types including int, short, long, float, double, and string.
2. Each class is responsible for marshalling and unmarshalling its own object. Each class inherits and implements the Serialize interface (created by us).

3. It is assumed that the entity classes defined in both the client and server use the same format and pattern for marshalling member variables in their objects.
4. The output of serialize functions and input of deserialize functions are char buffers.

3.1. Primitive data types

On the client side (C++), marshalling and unmarshalling of all primitive types is done using a generic function defined using C++ templates. On the server side (Java), marshalling of all primitive types is done using an overloaded function "serialize", whereas separate functions are defined for unmarshalling various primitive data types. These functions are implemented in the SerializePOD[1] files.

Marshalling is done by converting the input value to a char array and appending it to the target char buffer. For strings, the length of the string is encoded before the actual string value. For data types that occupy more than 1 byte, we convert the ordering from the host's ordering to Big-Endian ordering.

Unmarshalling is done by reading the required number of bytes from the input buffer and converting it into the target data type. The number of bytes to be read is determined by the size of the target data type (e.g., int – 4 bytes, float – 4 bytes, long – 8 bytes, etc.). For strings, the length of the string is decoded first. For data types that occupy more than 1 byte, we convert the ordering from Big-Endian ordering to the host's ordering.

3.2. Marshalling Client Request

The client request message has the following format:

Message Type	Request ID	Client IP	Content Size	Contents

1. Message Type: It has a value of either 0 (request message), or 1 (reply message). For client requests, it is set to 0. It is of type int.
2. Request ID: This is the ID of the request message sent by the client. It is of type int.
3. Client ID: This is the ID of the client sending the request. By default, it is the IP address + port number of the client. The client ID and request ID are used to keep track of message history for at-most-once server. It is of type string.
4. Content Size: The size in bytes of the message contents. It is of type int.
5. Contents: The actual message expressed as char buffer, for transfer over connection. It contains the service request type and the required arguments. It is of type string.

3.3. Unmarshalling Client Request

When a server receives a message over the communication module, following steps occur to process the message:

1. The request message is unmarshalled by calling deserialize method of Request class. This step unmarshalls the message type, request ID, client ID, and contents.
2. The contents are further unmarshalled to get service request type.
3. Forward the request to respective Skeleton class.

[1] SerializePOD stands for Serialize Plain Old Data

3.4. Marshalling Server Response

After processing a request, the skeleton instantiates a Reply message and serializes it. Server reply has the following format:

Message Type	Status	Content Size	Contents

1. Message Type: It has a value of either 0 (request message), or 1 (reply message). For server response, it is set to 1. It is of type int.
2. Status: It has a value of either 0 (success), or 1 (failure). It is of type int.
3. Content Size: The size in bytes of the message contents. It is of type int.
4. Contents: The actual message expressed as char buffer, for transfer over connection. It contains the results serialized as char buffer. It is of type string.

3.5. Unmarshalling Server Response

When a client receives a message over the communication module after making a request, following steps occur to process the message:

1. The reply message is unmarshalled by calling SerializePOD functions. This step unmarshalls the status, and contents.
2. If status is 0 (success), the contents are further unmarshalled to get the response object. The object is the displayed.
3. If status is 1 (failure), display the error message (string) received from server.

4. Services Provided

The following services are provided the Flight Information System: -

4.1. Query Flights Identifier(s) by specifying source and destination

This service allows the user to query all the flights available for the given source and destination. If no flight is available for the specified source and destination, an error message is displayed.

Input Parameters	String source, String destination
Returns	ArrayList<String> flightIdentifiers

Input Case	**Server Response**
Flight exists from the input source to input destination	Enter the source location Singapore Enter the destination location New Delhi Timeout : Did not receive anything from s erver... Retransmitting Message Num Flights: 2 AI381 UK116
Flight does not exist from the input source to input destination	Enter the source location Singapore Enter the destination location Australia No flights found

4.2. Query Flight Information

This service allows users to query the flight information including departure time, airfare and seat availability by specifying the flight identifier. If the flight with the requested identifier does not exist, an error message is displayed.

Input Parameters	String flightIdentifier
Returns	Flight object

Input Case	Server Response
Valid flight identifier	Flight Number : UK115 Source : New Delhi Destination : Singapore Departure Time : 11:10:00 Duration : 04:30 hr Available Seats : 175 Booked Seats : 55 Price : 240$
Invalid flight identifier	Enter the flight id UK990 Flight number does not exist

4.3. Make Seat Reservation

This service allows users to make seat reservations by specifying the flight identifier and number of seats to reserve. If successful, this service creates a new booking for the user and returns an acknowledgement of seat reservation and a unique booking identifier and. In case of incorrect user input (e.g., not-existing flight identifier or insufficient number of available seats), a proper error message is returned.

Input Parameters	String flightId, int numSeatsBooked
Returns	String bookingId

Input Case	Server Response
Valid flight identifier and number of seats to reserve < available seats	Booking ID : K6EDSA Flight ID : UK115 Customer Name : Srishti Number of Seats Booked: 2
Invalid flight identifier	Flight ID UK990 not found
Invalid Number of seats	Enter the flight ID UK115 Enter customer full name John Enter the number of seats to reserve -2 You should book atleast 1 seat
Valid flight identifier and number of seats to reserve > available seats	180 seats cannot be book on flight UK115. Only 175 are available.

4.4. Monitor Updates

This service allows a user to monitor updates made to the seat availability information of a flight at the server through a callback for a designated time period called monitor interval. To register, the client provides the flight identifier and the length of monitor interval to the server. On receiving the request to monitor updates, the server captures the IP address of the client, and adds the client to the list of subscribers. During the monitor interval, every time a **seat reservation is made by any client** on

the flight or if a **booking for this flight is cancelled by any client**, the **updated seat availability of the flight is sent** by the server to the registered client(s) through a callback. After the expiration of the monitor interval, the client record is removed from the server i.e., the client is unsubscribed from updates. For simplicity, we assume that the user that has issued a register request for monitoring is blocked from inputting any new request until the monitor interval expires.

An observer design pattern is used to implement the callback service. All the clients that are interested in monitoring updates act as observers and register their interest with the server. The skeleton of the monitoring service notifies the registered/subscribed clients whenever there is a change in the seat availability of their selected flight. This design pattern allows loose coupling between the observer (client) and the subject (monitoring skeleton). The diagram below illustrates this design.

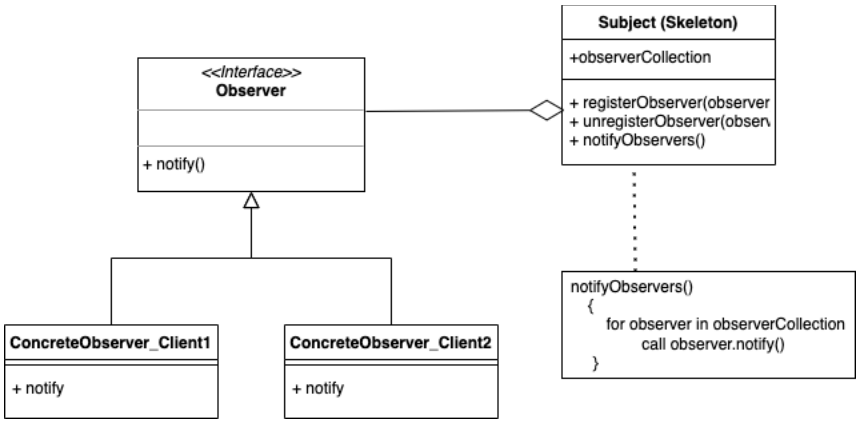

Figure 3. Observer design pattern

Input parameters	String flightId, int monitorInterval
Returns	String seatUpdateMsg

Input Case	Server Response
Valid flight identifier and monitor interval	Enter the Flight ID for monitoring seat updates EK354 For how many second(s) would you like to monitor this flight? 120 172.18.236.72:50142 subscribed to updates successfully
Invalid flight identifier	Enter the Flight ID for monitoring seat updates EK380 For how many second(s) would you like to monitor this flight? 34 Requested flight not found
Invalid Monitor Interval	Enter the Flight ID for monitoring seat updates EK354 For how many second(s) would you like to monitor this flight? -8 You should enter a positive number

Once the server has acknowledged that the client is subscribed, the client only **receives** messages till the end of the monitoring interval. This means that if any seat update data packet sent from the server is lost, the client will not send any request to retransmit.

4.5. Cancel Booking – Non-Idempotent Request

This service allows the user to cancel a reservation by providing the unique booking identifier. If successful, this service vacates the reserved seats and removes the user's records from the booking data store. If cancellation is unsuccessful due to an invalid booking id, an informative error message is displayed.

Input parameters	String bookingId
Returns	Boolean true/false

Input Case	Server Response
Valid booking identifier	Enter the booking ID (6 character) LK3W3C Booking LK3W3C cancelled succesfully
Invalid booking identifier	Enter the booking ID (6 character) PMY8IT Booking not found

4.6. Check Booking – Idempotent Request

This service allows the user to check the details of their flight booking by specifying the booking identifier. The output displays booking identifier, flight identifier, user's name and number of seats booked.

Input Parameters	String bookingId
Returns	Booking object

Input Case	Server Response
Valid booking identifier	Enter the booking ID (6 character) PJ7FR8 Booking ID : PJ7FR8 Flight ID : SQ305 Customer Name : Tommy S. Number of Seats Booked: 4
Invalid booking identifier	Enter the booking ID (6 character) MJ5DE9 Booking not found

4.7. Plan a Trip – Idempotent Request

This service takes the source and destination locations as input and returns the flight information of all possible routes to reach the destination from the source. In case there are no direct flights between source and destination, the service finds all possible connecting flight(s) to reach the destination. This service returns a list of all possible paths and the details of the trip including total cost, total travel time, flight identifier(s), flight timings, source & destination. We modelled the flights as a graph where the nodes are the locations, and a directed edge exists between nodes if a flight connects the pair of nodes. We used the Depth First Search algorithm to find all possible paths and we return the trip with short times and cheaper prices. The node labels illustrated in *Fig. 4* are as follows - New Delhi: 1, Mumbai: 2, Kolkata: 3, Dubai: 4, Muscat: 5, Singapore: 6, London: 7. This service is idempotent because we do not consider the availability of seats in the algorithm.

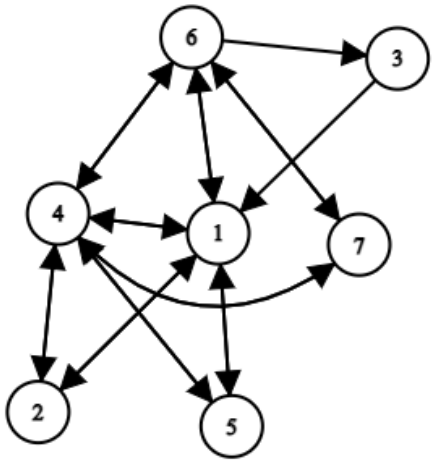

Figure 4. Graphical representation of flights and locations

Input parameters	String source, String destination
Returns	Trip [] trips

Input Case	Server Response
Paths Exist	Enter the source location Kolkata Enter the destination location Singapore Num Trips: 2 Price: 396$, Travel Time: 21:20:00 Kolkata(15:25:00) ---6E5078(03:30 hr)---> New Delhi(08:15:00) ---AI382(04:30 hr)---> Singapore(12:45:00) Price: 454$, Travel Time: 24:15:00 Kolkata(15:25:00) ---6E5078(03:30 hr)---> New Delhi(11:10:00) ---UK115(04:30 hr)---> Singapore(15:40:00)
No Path Exists	Enter the source location Australia Enter the destination location London No flights found

5. Fault Tolerance Experiments

In the following experiments we compared the two invocation semantics – at-least once and at-most once, by verifying their effectiveness on fault tolerance for idempotent and non-idempotent operations. For the experiments we simulated the loss of request and reply messages using probability thresholds of various values between 0 and 1. We used the **Query Flights Identifier(s) by specifying source and destination** (idempotent) and **Make Seat Reservation** (non-idempotent) as experiment queries to test fault tolerance.

The parameters passed for the two services were:

- **Query Flights Identifier(s) by specifying source and destination** - source: Dubai, destination: Singapore.
- **Make Seat Reservation** – name: Andrew Tate, flight_id: EK66, num_seats: 1.

The two operations were executed 1000 times and the expected value was compared with the actual value. We also reported the number of times the service was executed with both invocation semantics.

5.1. Simulating Loss of Request

In this experiment we set the probability of loss of request to values between 0 and 1 and set the probability of loss of reply to 0. This was done to simulate loss of only request and compare the fault-tolerance measures when the server does not receive the request from the client.

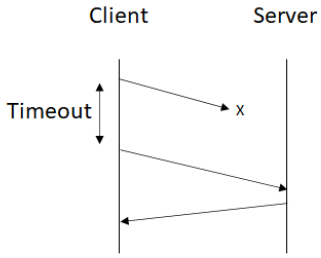

Figure 5. Simulation of request loss

Table 1. Request loss with at-most once invocation semantic on idempotent operations

Probability of loss of request	Expected Output	Actual Output	Number of Function Calls (avg)
0.9	SQ495 EK354	SQ495 EK354	1
0.5	SQ495 EK354	SQ495 EK354	1
0.1	SQ495 EK354	SQ495 EK354	1
0.05	SQ495 EK354	SQ495 EK354	1

Table 2. Request loss with at-least once invocation semantic on idempotent operations

Probability of loss of request	Expected Output	Actual Output	Number of Function Calls (avg)
0.95	SQ495 EK354	SQ495 EK354	1
0.9	SQ495 EK354	SQ495 EK354	1
0.5	SQ495 EK354	SQ495 EK354	1
0.1	SQ495 EK354	SQ495 EK354	1

Table 3. Request loss with at-most once invocation semantic on non-idempotent operations

Probability of loss of request	Expected Output (mean)	Actual Output (mean)	Number of Function Calls (avg)
0.95	449	449	1
0.9	449	449	1
0.5	449	449	1
0.1	449	449	1

Table 4. Request loss with at-least once invocation semantic on non-idempotent operations

Probability of loss of request	Expected Output (mean)	Actual Output (mean)	Number of Function Calls (avg)
0.95	449	449	1
0.9	449	449	1
0.5	449	449	1
0.1	449	449	1

5.2. Simulating Loss of Reply

In this experiment we set the probability of loss of reply to values between 0 and 1 and set the probability of loss of request to 0. This was done to simulate loss of only reply and compare the fault-tolerance measures when the client does not receive the reply from the server.

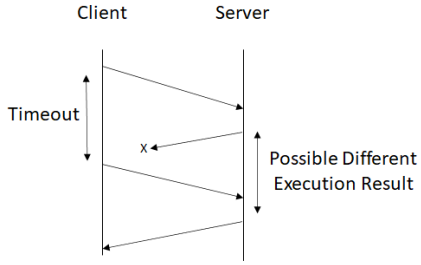

Figure 6. Simulation of reply loss

Table 5. Reply loss with at-most once invocation semantic on idempotent operation

Probability of loss of reply	Expected Output	Actual Output	Number of Function Calls (avg)
0.95	SQ495 EK354	SQ495 EK354	1
0.9	SQ495 EK354	SQ495 EK354	1
0.5	SQ495 EK354	SQ495 EK354	1
0.1	SQ495 EK354	SQ495 EK354	1

Table 6. Reply loss with at-least once invocation semantic on idempotent operation

Probability of loss of reply	Expected Output	Actual Output	Number of Function Calls (avg)
0.95	SQ495 EK354	SQ495 EK354	20.092
0.9	SQ495 EK354	SQ495 EK354	10.175
0.5	SQ495 EK354	SQ495 EK354	1.95
0.1	SQ495 EK354	SQ495 EK354	1.097

Table 7. Reply loss with at-most once invocation semantic on non-idempotent operation

Probability of loss of reply	Expected Output (mean)	Actual Output (mean)	Number of Function Calls (avg)
0.95	449	449	1
0.9	449	449	1
0.5	449	449	1
0.1	449	449	1

Table 8. Reply loss with at-least once invocation semantic on non-idempotent operation

Probability of loss of reply	Expected Output (mean)	Actual Output (mean)	Number of Function Calls (avg)
0.95	449	429.834	20.166
0.9	449	439.926	10.074
0.5	449	447.997	2.003
0.1	449	448.886	1.114

6. Inference

The two invocation semantics are robust to loss of requests. This is because for both invocation semantics, the client retransmits the request on timeout and the server will receive the request at some point in time (since probability (loss) < 1) and will return the result after execution. This is the same for both idempotent and non-idempotent operations and the function is executed only once in the server.

However, when there is a loss of reply the two invocation semantics behave different. The actual output is the same as expected output when using the at-most once server and the function call is executed only once for both idempotent and non-idempotent operations.

For the at-least once invocation semantic, the expected output is the same and the actual output for idempotent operations, but the number of function calls is greater than 1. When the probability is 0.5, the request gets executed approximately twice and when the probability is 0.9, the request is executed approximately 10 times. However, for the non-idempotent operation, the actual value differs from the expected value. When the probability is 0.5 approximately 2 seats are booked instead of 1 since the request gets executed 2 times and when the probability is 0.9, approximately 10 seats are booked. This shows that the at-least once server is unreliable when dealing with non-idempotent operations with reply losses.

The at-most once server can deal with all the cases and does not produce incorrect values in the data.