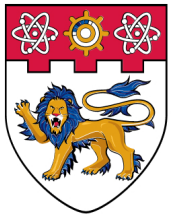


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE4013/CZ4013/SC4051

Distributed Systems:

Design and Implementation of A Distributed Flight Information System

Prepared by:

Lim Song Wei, Greg	U2120517G
Alloysius Lim Zong Hong	U1922302F
Hayden Ang	N2203300H

1. Introduction

The purpose of this project is to design and develop a flight information system for communicating between clients and a server, comparing the difference between different invocation semantics, namely, at least once and at most once invocation.

As part of the requirements, we have to implement our own communication protocol using only UDP, ensuring reliable communication between poor network conditions. Our protocol should serialize and deserialize the data for communication. The client must be able to register a callback to be notified of changes in the number of flight seats.

Both invocation semantics must implement relevant features to ensure successful communication such as message timeout and re-sending. The at-most once invocation must include message caching to prevent double execution and replying to duplicate requests.

1.1 Environment

The following are the environments required to run the flight information system:

1. Programming Language - The client and server software were developed using Java 19 and Apache Maven project build tools.
2. Operating System - The application was developed and tested in Windows 11.

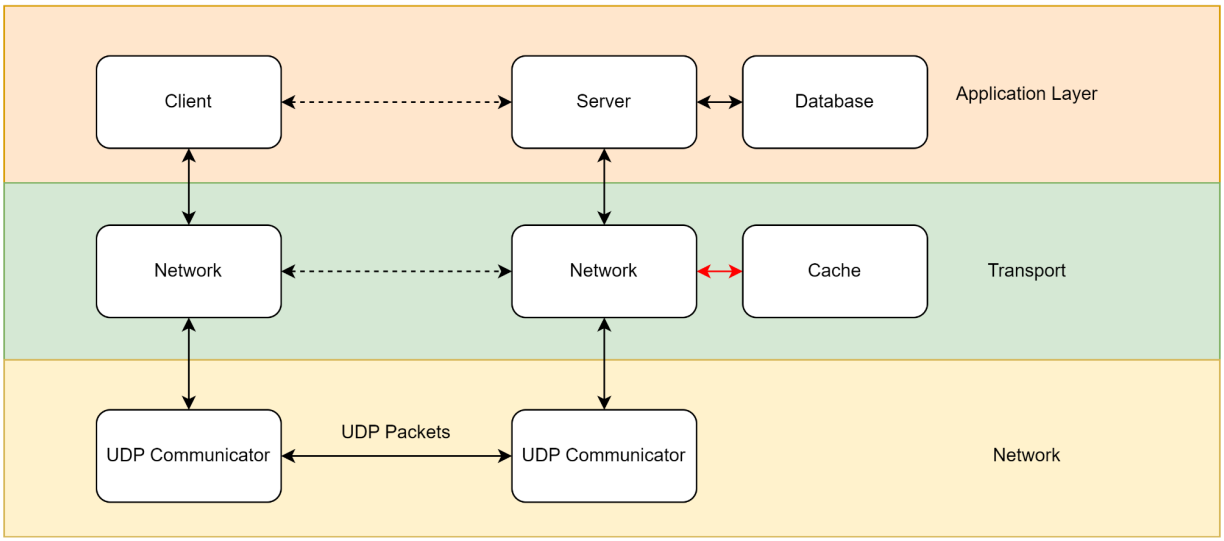
1.2 Assumption

The assumptions made while we develop the system are as stated:

- User Interface
    - The user interface is entirely built in the command-line interface(CLI).
- Request Concurrency
    - Requests by clients are assumed to be made well separated in time so that before finishing processing a request, the server will not receive any new request from any client.
- Client-Server Communication
    - The server address and port number are assumed to be known by the client.
- Storage
    - The server stores all information in memory.

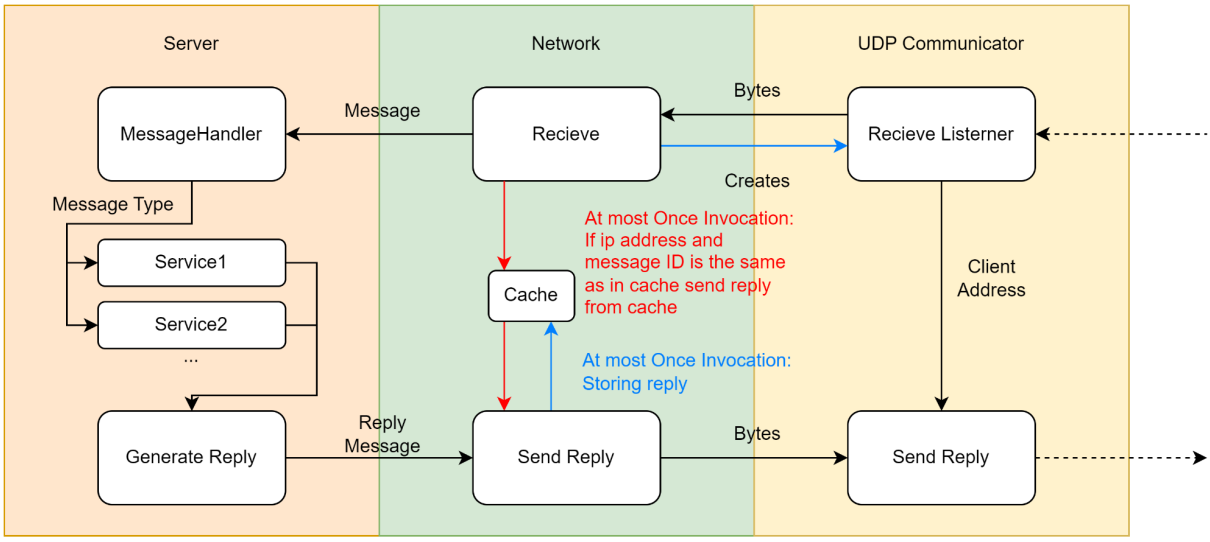
2. Design

2.1 Architecture Design



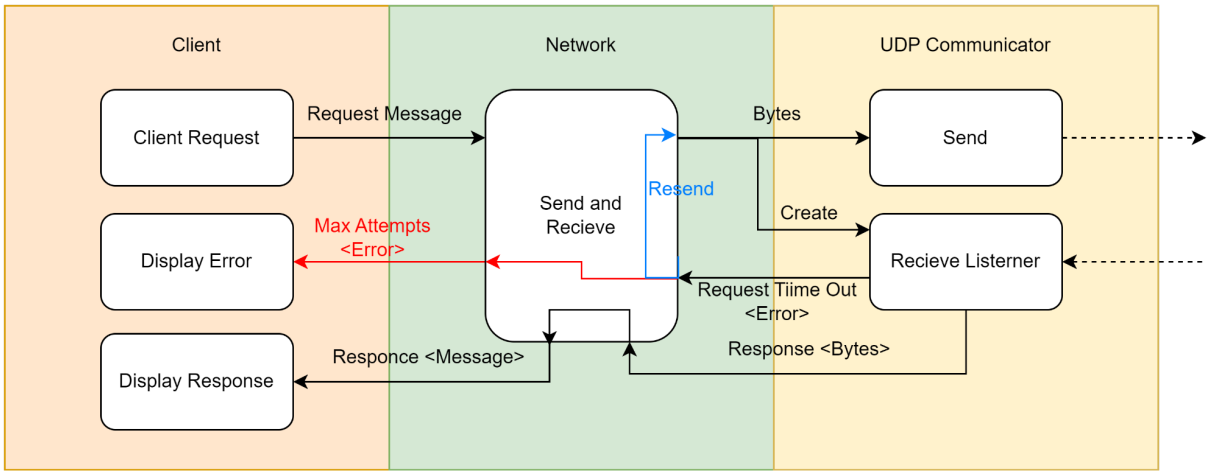
The figure shows an overall view of how different aspects of the entire application communicate. For the client and server, all messages are passed by the UDPCommunicator. However, from the network, Client and Sever perspective, they are communicating directly with all complexity handled by the layer below.

Server



The server requests the network to create a Recieve Listerner to wait for a message. The bytes are unmarshall to a Message format. Depending on the invocation, a cache might be used. The message is passed to the Server to deserialize the Message depending on the message type. The message is then passed to the Network to send reply and cache dependingly.

Client



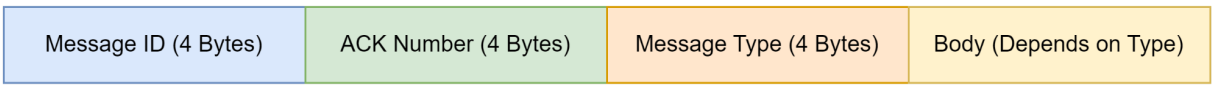
Our client begins with a user inputs to generate a Request message. The Message Type is determined by the User and its body serialize based on the Type. The Server uses a network method "SendAndRecieve" to handle automatic resending of packets if it has not received any reply. The method send a message and creates a Recieve Listerner with a give timeout. On time out, the method would send the packet again. When the max attempts is reached, it return a Error. When a response is received, it passes the massage back to the Client.

2.2 Communication Design

Message Structure

Both Client and Server use the same message format. This message is serialized by Network before Sending and Desirialise when receiving.

Message format



Message ID:

The ID is unique for each request from the Client or server. Same ID and IP implies the same body.

ACK Number:

The ACK Number is to inform the recipient which request this message is replying to.
The Reply ACK Number is always request MessageID+1

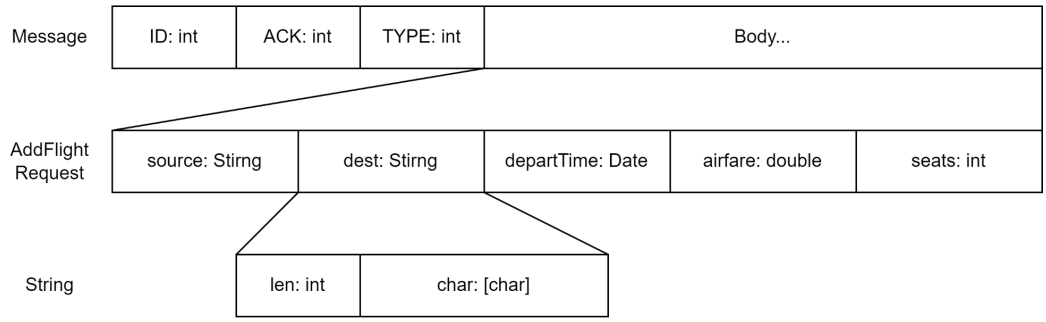
Message Type:

Value 1 - 7 are request
Value 11 - 17 are response
Value 999 is an error type
Other values are for callback messages

Body:

Just bytes, to be deserialized depending on the message type.
If the type is error (999) the body is just a string with error message
The respective Client and Server handlers will Deserialise accordingly

Message Body Structure



There are different types of request and reply format messages. These formats are at "entity/messageFormats"; all of these format can be marshaled and unmarshalled. Additionally, there are some additional types such as DateTime and FlightInfo as common formats used by other formats.

Marshaling and Unmarshalling

There are 3 key types of data types that are taken into account, primitive, custom request and reply, and special formats. All formats are in big endian.
For primitives, it is handled by "MarshallUtils". Date is handled here and is stored as an Int. Strings are also primitive types and are stored in 4 bytes for the length, followed by the bytes.
Custom formats each have their own marshaling and unmarshalling methods. All requests and replies except for errors use this format.
Lastly, special formats like "List<FlightInfo>" have a custom helper function for other messages to use.

3. Services

Service 1: Get Flight Detail Using Src and Dest

This service allows users to search for all flights corresponding to a given source country and destination country. If the query is successfully processed, the server replies with a response containing a list of all Flights that have the corresponding source country and destination country.

The request message of this service has the following format:

Content	Parameter	Type
1	Source Country (src)	String
2	Destination Country (dest)	String

The response message of this service has the following format:

Content	Parameter	Type
1	FlightInfoList	List<FlightInfo>

Service 2: Get Flight Detail Using ID

This service allows users to search for a particular flight by specifying a Flight ID. If the query is successfully processed, the server replies with a response containing a list containing the Flight that has the corresponding Flight ID.

The request message of this service has the following format:

Content	Parameter	Type
1	FlightID(id)	int

The response message of this service has the following format:

Content	Parameter	Type
1	FlightInfoList	List<FlightInfo>

Service 3: Reserve Seats Using ID and Number of Seats

This service allows users to reserve a number of seat(s) for a particular flight by specifying the Flight ID and the desired number of seats. If the service is successfully processed, the server replies with a response containing a message informing the customer that the desired number of seats has been reserved.

The request message of this service has the following format:

Content	Parameter	Type
1	FlightID(id)	int
2	NumberOfSeats(noSeats)	int

The response message of this service has the following format:

Content	Parameter	Type
1	Message	String

When this service is called, the server would request for all valid callbacks with the same flight ID from the Database and send a message to notify the client. Note that the server only sends the notification once with format.

Content	Parameter	Type
1	Acknowledgement Message	String

Service 4: Set Notification for when Number of Seats changes

This service allows users to set a notification to inform them when the number of seats changes for a particular flight over a specified duration of time. If the service is successfully processed, the server replies with a response containing a message informing the user that the notification has been set.

The request message of this service has the following format:

Content	Parameter	Type
1	FlightID(id)	int
2	callBackDuration	int

On Successful adding of callback, the client receives:

Content	Parameter	Type
1	Acknowledgement message	String

The client then create an Recieve Listerner for the duration of the call back timer.

Service 5: Set Flight Price

This service only allows authenticated users with the correct SessionID(key) to set/change the price of a particular flight by specifying the Flight ID and the desired price of the flight. If the service is successfully processed, the server replies with a response containing a message informing the customer that the price of the flight has been changed to the desired price.

The request message of this service has the following format:

Content	Parameter	Type
1	FlightID(id)	int
2	FlightPrice	double
3	SessionID(key)	int

The response message of this service has the following format:

Content	Parameter	Type
1	Acknowledgement Message	String

Service 6: Create a New Flight

This service allows users to create a new flight by specifying the necessary flight information. If the service is successfully processed, the server replies with a response containing a message informing the user that the new flight has been created along with the UID generated by the server that has been assigned to the new flight. Note that this method is non-idempotent, as the server generates an UID for each new flight request.

The request message of this service has the following format:

Content	Parameter	Type
1	Source Country (src)	String
2	Destination Country (dest)	String
3	DepartureTime	DateTime
4	FlightPrice	double
5	NumberOfSeatsAvailable	int

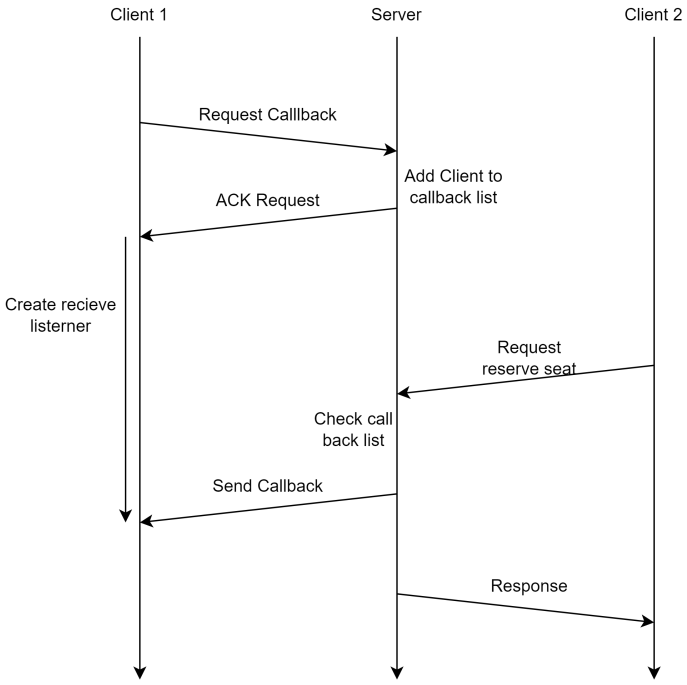
The response message of this service has the following format:

Content	Parameter	Type
1	Ack Message	String

Callback of Service 4 when service 3 is executed

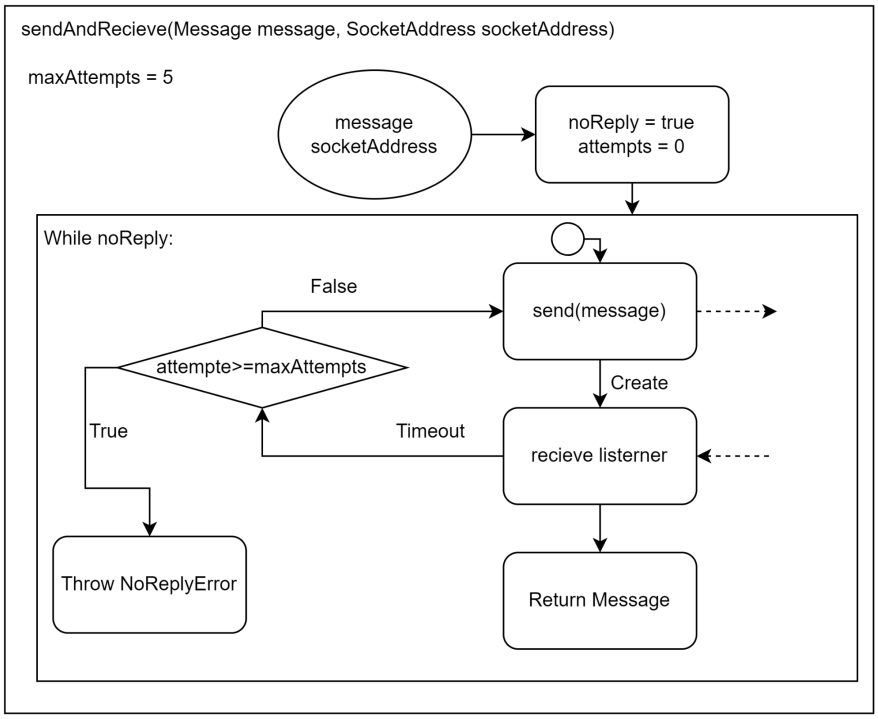
The diagram shows how Clients and Server handles callback functions. A client sends a request to the server to track changes to the number of seats on a flight. If the client request is acknowledged, the client will set up a receive listener to wait for the callback. The server would send a call back if the number of seats on the specific flight changes.

Note that the callback packet may be lost requiring the client to timeout. This is a limitation of our implementation.



4. Fault Tolerance

4.1 Message resending



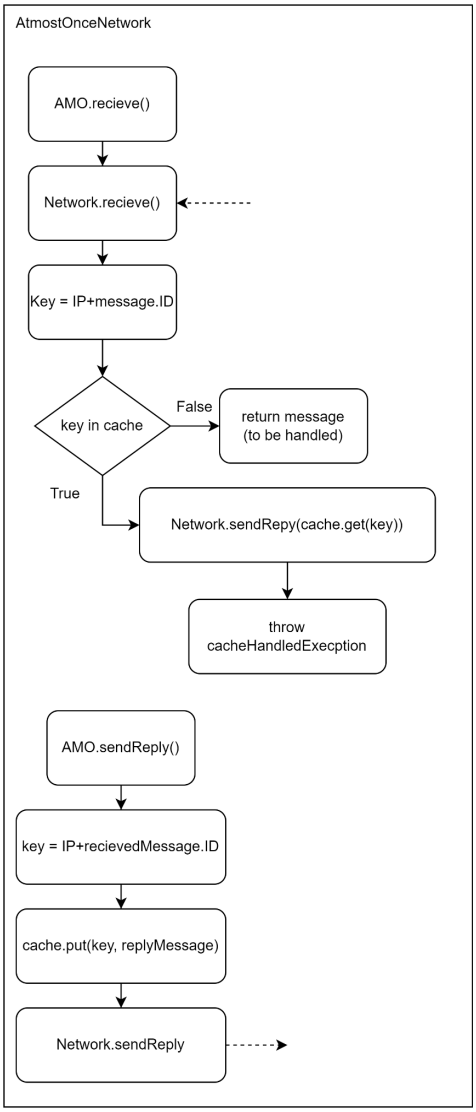
As part of fault tolerance, message resending is required to ensure successful communications. For both invocation, the Netwrok Method "sendAndRecieve will keep sending till it gets a response or the maxAttempt is reached and it throws an error to the Client. This method will keep sending request till it receives a response or it reach its max attempts.

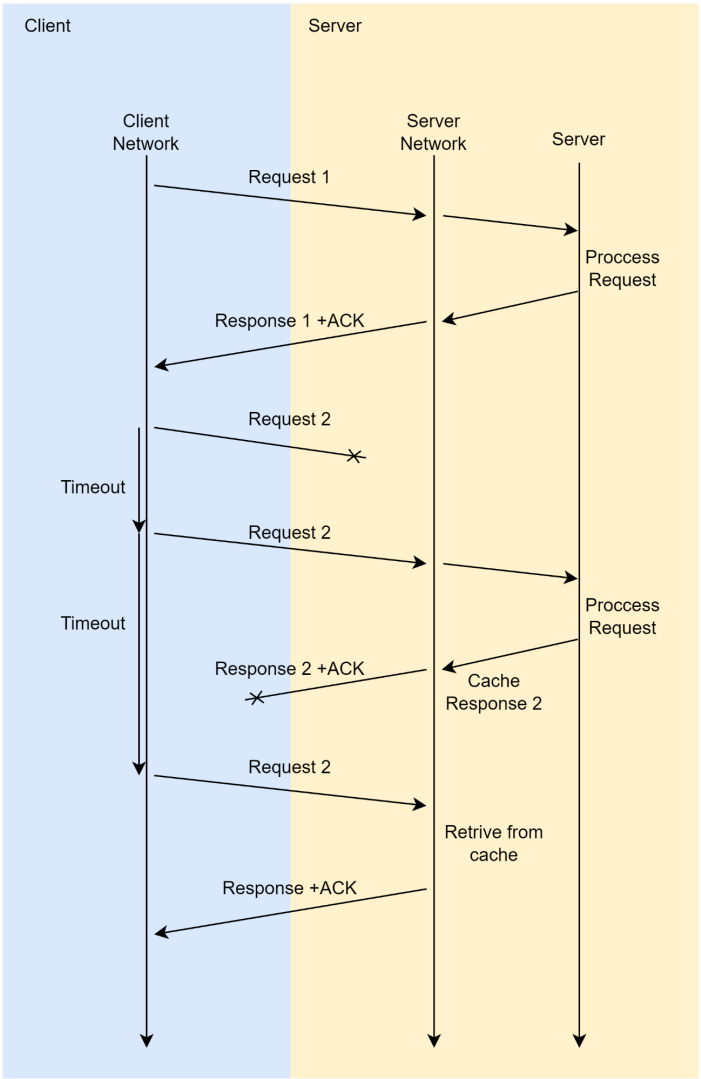
4.2 At-Least Once Invocation

In at-least once invocation, no additional work is required as adding the auto resending of messages and the server executing all requests it receives has achieved the requirement of at least once.

4.3 At-Most Once Invocation with caching

In at-most once invocation, the sending of messages via "sendAndRecieve" method does not need any changes.





The left diagram shows changes that are required for AtMostOnce Invocation. The "receive" and "sendReply" need to be overwritten and an additional cache is added to "AtmostOnceNetwork" that extends "Network". The cache is a dictionary that maps (IP+messageID) to messages that it has been sent.

"AtmostOnceNetwork.recieve()" check every message for duplication against the cache. If there is a duplicate the network would send a reply using the cache and throw a "cacheHandledException" to the server to inform it.

"AtmostOnceNetwork.sendReply()" caches every reply it sends. As this method is called when the server wants to reply to a request, this reply is stored in the cache for all future messages it receives.

The right diagram shows all possible message exchanges for AtMostOnce Invocation. Message 1 is when no packets are lost. For message 2, the first client side packet is lost. The client timeout and resends the request. The server's network receives the message, checks the cache for duplicate messages, and passes it to the server for processing. The server passes the reply back to the network to be sent. Unfortunately, the server response 2 packet is lost. The client timeout and resend a request. The server network checks the cache and finds a duplicate. The server network retrieves the cache response and sends it back to the client with out re-proccessing the request.

4.4 Comparison

Idempotent

In our program, requesting flight ID, requesting flight information and changing airfare are idempotent requests.
Monitoring of seats is not strictly idempotent as if the server receives multiple requests to register a callback for a client, multiple of these callbacks are stored; however, since duplicate callbacks have the same effects, there is no problem with the server storing identical callbacks.

Idempotent request	At-Least Once	At-Most Once
No Fault	Client receive reply	Client receive reply
Client Packet Lost	No response, Client resend	No response, Client resend
Server Packet Lost	Server process the request but reply packet lost Client resends, Server re-process	Server process the request and caches reply but reply packet lost Client resends, server use cache and send reply
Client Max Attempts	Client Sends multiple times with no response. Server may execute requests multiple times (waste resources). Client stops sending (maybe server down or poor network)	Client Sends multiple times with no response. Server will only execute once. Client stops sending (maybe server down or poor network)

Although duplicate request and multiple execution of idempotent requests does not affect the state of the server and database more than once, if its execution is resource intensive, it might affect the server performance for multiple executions and generating repeated responses.

Non-Idempotent

In our program, there are 2 non-idempotent requests, namely making flight reservations and creating new flights.
Making a flight reservation is non-idempotent and the number of seats available in a flight after a successful reservation is made decreases. If multiple requests are processed, the remaining seats will be decreased multiple times.

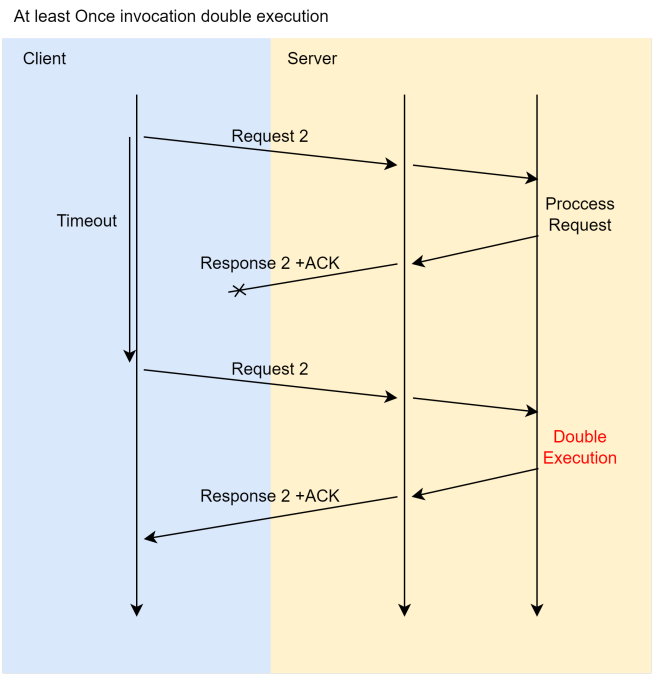
Create new flight is non-idempotent as flight IDs are generated by the server per request, and it is assume that there can be 2 flights with identical information but different ID (ID→FlightInfo, FlightInfo↛ID). This means that multiple requests with the same flight information will result in multiple flights with different ID being created.

Idempotent request	At-Least Once	At-Most Once
No Fault	Client receive reply	Client receive reply
Client Packet Lost	No response, Client resend	No response, Client resend
Server Packet Lost	Server process the request but reply packet lost Client resends, Server re-process	Server process the request and caches reply but reply packet lost Client resends, server use cache and send reply
Client Max Attempts	Client Sends multiple times with no response. Server may execute requests multiple times resulting in an incorrect database state. Client stops sending (maybe server down or poor network)	Client Sends multiple times with no response. Server will only execute once. Client stops sending (maybe server down or poor network)

For at-least once invocation of non-idempotent request, the server will process the same request multiple times while the client will only get one response resulting in wrong results on the server.

The diagram shows how a double execution might occur when using at-least once invocation. For example, when performing seat reservation, the client sends "Reserve 3 seats on flight 2020" the server executes but the reply packet is lost. The Client timeout and resends the request. The server receives again and executes again. This results in 6 seats being reserved.

For at-most once, it is not possible to double executed as all repeat request are handled by the network and not passes to the server.



5. Conclusion

In this project, we design an implemented serializing and deserializing of data for network sending. Designed our own message and message body formats. Implemented message resending to improve the reliability over a poor network. Added at-least once and at-most once network invocation to handle duplicate requests. From implementing these strategies, we learned about issues of at least once invocation such as double execution and for non-idempotent requests, wrong server results. Lastly, we design and implement a callback when the number of seat reserved changes.