NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

School of Computer Science and Engineering

CX4013: Distributed Systems

Flight Booking System

Project Demo Session: 4 April 2023, 1.45pm to 2.00pm

Group Members:

Seenivasan Sashwath Ravilla (U1922881F) (33%)

Lim Wi Teow (U1921765G) (33%)

Yap Zher Hao, Jonathan (U1922422K) (33%)

Program Summary

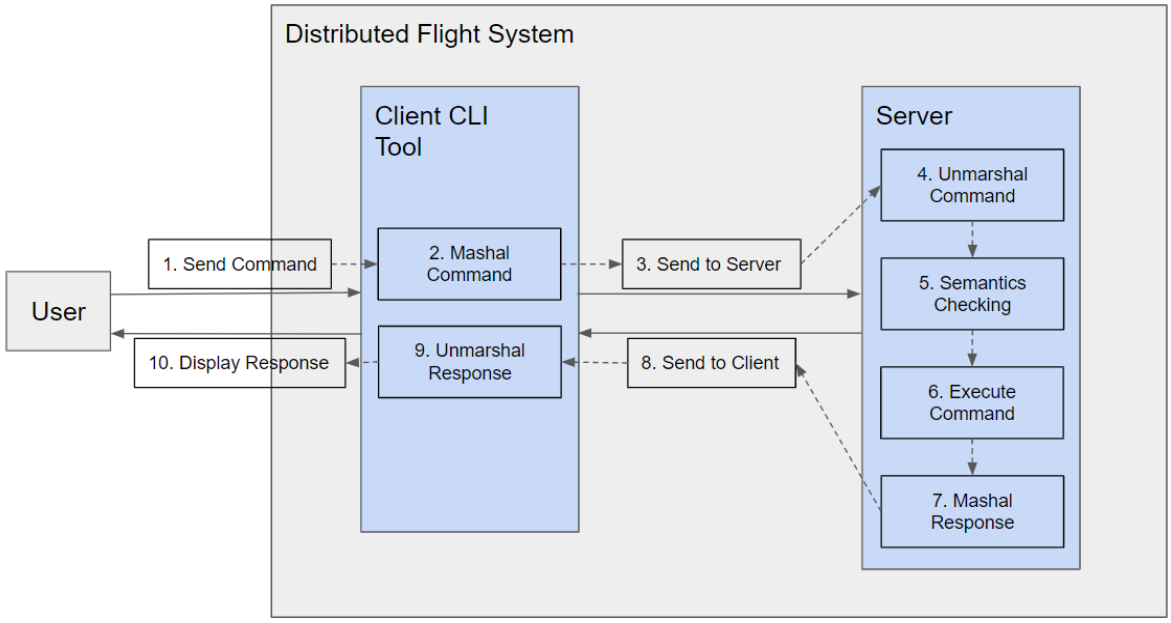
The objective of this project is to design and implement a simple flight booking system using a client-server architecture. The client component will be developed using Python, while the server component will be developed using Java.

Choice of Programming Languages

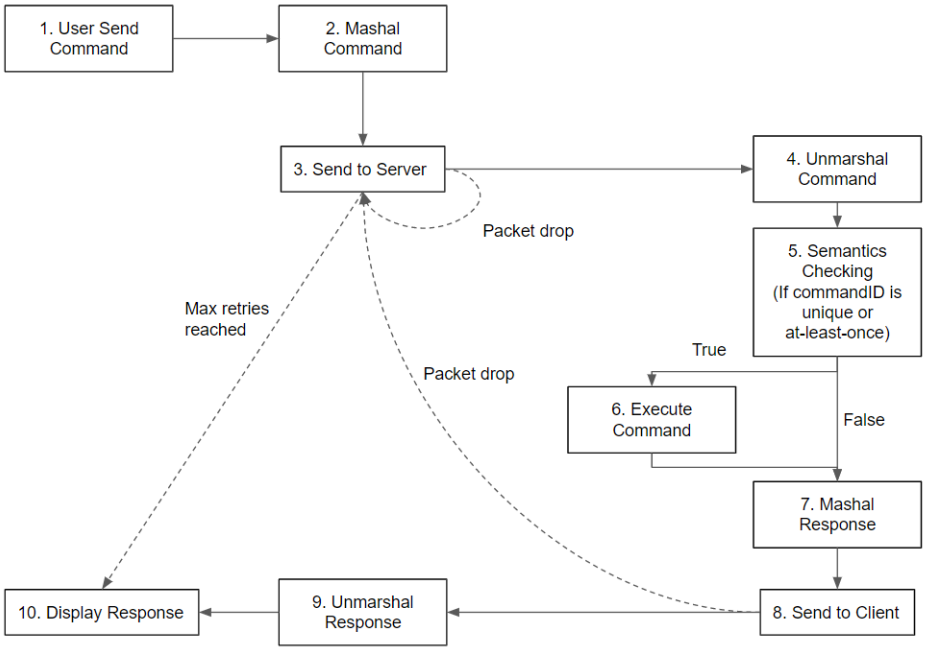
Python was chosen for the client program due to its simplicity and ease of use, while Java was chosen for the server program for its maturity, robustness, concurrency, scalability, and performance. Python's dynamic typing and interpreted nature make it quick to write and test code, but Java's compiled nature is optimized for speed and performance. Using different languages for the client and server programs allows developers to take advantage of the strengths of each language and build a system that is optimized for both client-side and server-side tasks, resulting in better performance, scalability, and maintainability.

Program Flow

Architecture Diagram



Control Flow Diagram



The client command line tool is the only point of entry into the system for the user. Once a command is inputted, it undergoes processing and marshalling to prepare it for transmission to the server. Subsequently, the client program establishes a connection with the server by opening a port and transmitting the marshalled command. Upon receipt, the server unmarshals the command and executes it, after which it sends the results back to the client.

In the event of packet loss, the client will wait for a specified timeout duration before retransmitting the exact same command, complete with the same command ID. This mechanism allows the server to determine whether it has executed the command previously and respond according to the preconfigured semantics.

For the callback function, the client sets a monitoring interval and opens the corresponding port for incoming information. The server then sends updates to the client through the open port as necessary. Upon completion of the monitoring interval, the client sends a command to inform the server that the monitoring has ended.

Marshalling

```python
def marshal(unmarshalled_data):
  # marshal logic
  marshalled_data = bytearray()
  for key, value in unmarshalled_data.items():
      key_len = str(len(key.encode('utf-8')))
      if len(key_len) == 1:
        padded_key_len = "00" + key_len
      elif len(key_len) == 2:
        padded_key_len = "0" + key_len
      elif len(key_len) == 3:
        padded_key_len = key_len
      marshalled_data += padded_key_len.encode('utf-8')
      marshalled_data += key.encode('utf-8')

      value_len = str(len(value.encode('utf-8')))
      if len(value_len) == 1:
        padded_value_len = "00" + value_len
      elif len(value_len) == 2:
        padded_value_len = "0" + value_len
      elif len(value_len) == 3:
        padded_value_len = value_len
      marshalled_data += padded_value_len.encode('utf-8')
      marshalled_data += value.encode('utf-8')
  return marshalled_data
```

Data marshalling is performed before data is sent out from both the client and server. When a request is being sent from the client to the server, the client will format the message in a specific way. A 3-digit integer is used to denote the length of the proceeding message. For example, if the message is "command", the 3-digit integer "007" will be appended before it to indicate that the message is 7 characters long. The lengths of the messages are not fixed and thus it is important to indicate the length of the message.

Additionally, by specifying the length of the message, it is possible to detect errors in transmission, such as missing or extra data, which can be crucial for ensuring data integrity and reliability.

```java
//allocate first 3 bytes for length
for (int i = 0; i < MAX_INT_LENGTH; i++) {
      lenBuilder.append((char) inByteArr[bytesPos]);
    lenBuilder.append((char) byteList.remove( index: 0).byteValue());
    bytesPos += 1;
}
int keyLength = Integer.parseInt(lenBuilder.toString());
System.out.println("keyLength: " + keyLength);

// find string of key
while (keyLength > 0) {
    keyBuilder.append((char) byteList.remove( index: 0).byteValue());
    keyLength -= 1;
    bytesPos += 1;
}
String keyString = keyBuilder.toString();
System.out.println("keyString: " + keyString);
lenBuilder.setLength(0);
keyBuilder.setLength(0);
```

```java
//allocate first 3 bytes for length
for (int i = 0; i < MAX_INT_LENGTH; i++) {
    lenBuilder.append((char) byteList.remove( index: 0).byteValue());
    bytesPos += 1;
}
int valLength = Integer.parseInt(lenBuilder.toString());
System.out.println("keyLength: " + valLength);

while (valLength > 0) {
    valBuilder.append((char) byteList.remove( index: 0).byteValue());
    valLength -= 1;
    bytesPos += 1;
}
String valString = valBuilder.toString();
System.out.println("keyString: " + valString);

unmarshallMap.put(keyString, valString);
valBuilder.setLength(0);
lenBuilder.setLength(0);
if (keyString.equals("command")) {
    commandValue = valString;
    break;
}
```

Similarly on the server side, the unmarshalling of the request will be the opposite of how the marshalling is done on the client side, hence reading the first 3 characters to be the length of the string while the subsequent string with the specified length will be the string itself.

Depending on the different client request, the server will then be able to know the data type of the client input to expect and hence unmarshall the input correspondingly.

Semantics

```
public UserInfo(String ipAdd, Integer localPort, String semantics){
    this.ipAdd = ipAdd;
    this.semantics = semantics;
    this.localPort = localPort;
    // this.responseId = new ArrayList<String>();
    this.responseList = new HashMap<String, String>();
    this.callbackFlight = "";
}
```

In this project, we have implemented 2 different invocation semantics, at-least-once and at-most-once. The logic of checking the semantics will be at the server side. Since there will be different UserInfo objects created for different clients, each object will have a semantics field that specifies the semantics settings for the particular user.

```
// check semantics
// execute request if semantics is at-least-once
if (serverController.checkSemantics(requestQuery, currUser)){
    // if not duplicate request, handle request
    // save the request and its response
    response = handleRequest(requestQuery, currUser);
    currUser.setResponse(requestQuery.toString(), response);
    System.out.println("Response generated as " + response.toString());
}
else{
    // if at-most-once request
    response = currUser.getResponse(requestQuery.toString());
    // generate reponse if not generated before
    if (response == null){
        // System.out.println("Duplicate request, sending stored info...");
        response = handleRequest(requestQuery, currUser);
    }
    currUser.setResponse(requestQuery.toString(), response);
}
```

After unmarshalling the client request, the server will check the semantics set for the client, with the default to be “at-most-once” for new clients. In at-least-once invocation semantics, the request message can be retransmitted, but there is no filtering of duplicate requests. In our implementation, duplicate requests received by the server will be executed.

However, for at-most-once invocation semantics, the request message can be retransmitted, and there is filtering of duplicate messages. When a request is received by the server, it is first checked if the request is a duplicate by running getResponse(), and it will resend the reply that has been sent before.

Client

cli.py

This Python script contains functions that make API calls to a flight reservation system and handles command-line arguments to determine which API call to make and with what parameters. Additionally, there are functions for reading and writing to a configuration file that stores the IP address, port number, and semantics of the flight reservation system.

The script's functions include *read_config()*, *write_config()*, *load_config()*, *increment_commandID()*, *queryID()*, *queryDetails()*, *reserve()*, *subscribe()*, *retrieve()*, *cancel()*, and *config()*. These functions handle tasks such as calling specific API functions, updating the configuration file, and printing the results of API calls. The script also utilises standard library modules such as argparse, sys, json, and pathlib, and custom modules.

api.py

This Python file contains functions for sending and receiving data between the Python client and Java server using UDP sockets. There are several functions defined, such as *sendRequest()*, *sendToJava()*, and *server_program()*, each with its specific purpose. There are also functions for calling different APIs, such as *queryID()*, *queryDetails()*, *reserve()*, *subscribe()*, *retrieve()*, *cancel()*, and *setSemantics()*, which format data in a dictionary, marshal it, and then send it to the Java server, returning the unmarshalled data. Additionally, there are constants defined for the maximum number of retries, the retry delay, and the maximum packet size.

marshalling.py

This python file defines two functions, *marshal()* and *unmarshal()*, that convert data between a dictionary format and a byte format. Information regarding the actual implementation has been mentioned earlier.

Server

The server is broadly categorised into 3 main packages:

Package	Description
Entity	Contains objects like BookingInfo, FlightInfo, UserInfo to be instantiated and used while the server is running
Marshalling	Contains the Marshaller object to marshal response and unmarshal requests.
Server	Contains server related files including serverEntity that is the server object, serverController that performs all the program execution and serverDatabase where the flights are initially generated

While the server is running, there are also static variables instantiated so as to keep track of the current state of each object (BookingInfo, FlightInfo, UserInfo). These static variables, which are instantiated in serverDatabase.java, are as follows:

Variables	Description
flightInfoArrayList	ArrayList containing all FlightInfo objects instantiated when the server starts.
userInfoArrayList	ArrayList containing all UserInfo objects whenever a client communicates to the server.
bookingInfoArrayList	ArrayList containing all BookingInfo objects whenever a client books a flight in the server.
callbackHmap	Hashmap with the key as a String and the value as an ArrayList of UserInfo. This is used to send updates to the client who subscribed to a particular flight Id (being the String).

Services

The flight booking system provides a wide array of services for the user to use. From serverEntity.class, there will be a switch case for different services which will call their respective functions from the serverController.class. These services are differentiated by their command ID when the request is being sent over (second string in every query).

Query the flight identifier(s)

This service allows users to query the flight identifiers for all flights by specifying the source and destination. If the request is successfully processed, the server replies with a response of the

flight identifier of all flights on that specific route. The command ID for this service is “queryID”. This service is idempotent as the request will not cause a change in the flight information.

Query	dfs queryID SINGAPORE CHINA
Response	FlightID: 2304041000 Source: SINGAPORE Destination: CHINA FlightID: 2304131315 Source: SINGAPORE Destination: CHINA

Query the departure time, airfare and seat availability

This service allows the user to query the flight information for a flight by specifying its identifier. If the request is successfully processed, the server replies with a response of departure time, airfare, and seat availability for that specific flight. The command ID for this service is “queryDetails”. This service is idempotent as the request will not cause a change in the flight information.

Query	dfs queryDetails 2304131315
Response	Departure Time: Thu, 13/Apr/2023 - 1:15 pm Airfare: $280.25 Seats Availability: 25

Make a seat reservation

This service allows the user to make a seat reservation on a flight by specifying the flight identifier and the number of seats to reserve. If the request is successfully processed, the server replies with an acknowledgement, booking ID, number of seats booked, and other relevant flight information. The command ID for this service is “reserve”. This service is non-idempotent as each request will reduce the number of available seats on a flight.

Query	dfs reserve 2304131315 1
Response	Booking ID: 1679979926961 Flight ID: 2304131315 Source: SINGAPORE Destination: CHINA Departure Time: Thu, 13/Apr/2023 - 13:15 pm Total Airfare: 280.25 Number of seats booked: 1

Monitor updates made to flight information (Callback)

This service provides a subscription to callback monitoring of a flight by specifying the flight ID and monitor interval. If the request is successfully processed, the server replies with an

acknowledgement. Subsequently, the user will be updated if there is any change made to the flight being monitored. The command ID for this service is “subscribe ”. This service is idempotent as the request will not cause a change in the flight information.

Query	dfs subscribe 2304131315 1
Response	Subscribe Request Sent! Waiting for 1 minutes... Response received Flight ID 2304131315 is current being monitored for 1 minutes...

When the flight information had been updated:

Response	Connection from: ('192.168.188.80', 51074) The following flight information has been updated: Flight ID: 2304131315 Source: SINGAPORE Destination: CHINA Departure Time: 2023-04-13T13:15 Airfare: $280.25 Available Seats: 22

After monitoring the flight for a specified duration, another request will be sent from the client to the server to stop the callback. This will unsubscribe the client from receiving any flight updates and removing the client from the ArrayList tagged to the flight ID in *callbackHmap*. The command ID for this service is “cancelCallback”, although the client will automatically send this request instead.

Query	dfs cancelCallback 2304131315
Response	Monitoring has stopped

Retrieve booking information

This service is one of the self designed services that is idempotent. This service allows the user to retrieve the booking information of a booking that has been made previously by specifying the booking ID. If the request is successfully processed, the server replies with the booking ID, number of seats booked, and other relevant flight information. The command ID for this service is “retrieve”. This service is idempotent as the request will not cause a change in the flight information.

Query	dfs retrieve 1679981764331
Response	Booking has been confirmed for the following: Booking ID: 1679981764331 Flight ID: 2304131315 Source: SINGAPORE

	Destination: CHINA Departure Time: Thu, 13/Apr/2023 - 1:15 pm Total Airfare: 280.25 Number of seats booked: 1

Cancel booking reservation

This service is one of the self designed services that is non-idempotent.This service allows the user to cancel a booking reservation that had been made previously by specifying the booking ID. If the request is successfully processed, the server replies with an acknowledgement and the details of the booking that was cancelled. The command ID for this service is “cancel”. This service is non-idempotent as a request will remove a booking from the server.

Query	dfs cancel 1679981552868
Response	Response received Booking has been cancelled for the following: Booking ID: 1679981552868 Flight ID: 2304131315 Source: SINGAPORE Destination: CHINA Departure Time: Thu, 13/Apr/2023 - 1:15 pm Total Airfare: 280.25 Number of seats booked: 1

Change semantics

Although this is not part of the requirements, we have included this service to experiment between 2 different semantics invocations. This service allows the user to change the invocation semantics by specifying “1” for at-least-once, or “2” for at-most-once. If the request is successfully processed, the server replies with an acknowledgement and the updated invocation semantics. The command ID for this service is “config”. This service is non-idempotent as a request will change the type of invocation semantics.

Query	dfs config -s 1

Response	Response received
	Semantics changed: at-least-once
	{
	"serverIP": "192.168.188.80",
	"serverPort": "12345",
	"semantics": "at-least-once",
	"commandID": "24"
	}

Comparison of Invocation Semantics

Making a seat reservation

With at-least-once semantics, a request may be executed twice if the initial reply message has been lost. This would lead to wrong results for a non-idempotent operation such as making a seat reservation.

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs reserve 2304131315 1
{'id': '7', 'command': 'reserve', 'flightID': '2304131315', 'noOfSeats': '1'}
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id0017007command007reserve008flightID010230413131500 9noOfSeats0011')
Sending Request
Message Sent!
Response received
Booking has been confirmed for the following:
Booking ID: 1679980047507
Flight ID: 2304131315
Source: SINGAPORE
Destination: CHINA
Departure Time: Thu, 13/Apr/2023 - 13:15
Total Airfare: 280.25
Number of seats booked: 1
```

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs queryDetails 2304131315
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id00210007command012queryDetails008flightID0102304131315')
Sending Request
Message Sent!
Response received
Departure Time: Thu, 13/Apr/2023 - 1:1547700000
Airfare: 280.25
Seats Availability: 24
```

When a reservation is made for 1 seat, the seat availability changes from 25 to 24.

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs queryDetails 2304131315
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id00212007command012queryDetails008flightID0102304131315')
Sending Request
Message Sent!
Response received
Departure Time: Thu, 13/Apr/2023 - 1:1547700000
Airfare: 280.25
Seats Availability: 23
```

However, if the response for the request to make a reservation has been dropped, the request would have been retransmitted. This would result in the request being executed twice, causing the seat availability to change from 25 to 23, therefore producing a wrong result.

With at-most-once semantics, the reply is stored, and the stored reply is retransmitted in the event of the request being lost.

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs reserve 2304041000 1
{'id': '15', 'command': 'reserve', 'flightID': '2304041000', 'noOfSeats': '1'}
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id00215007command007reserve008flightID0102304041000009noOfSeats0011')
Sending Request
Message Sent!
Response received
Booking has been confirmed for the following:
Booking ID: 1679980558796
Flight ID: 2304041000
Source: SINGAPORE
Destination: CHINA
Departure Time: Tue, 04/Apr/2023 - 10:00
Total Airfare: 300.0
Number of seats booked: 1
```

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs queryDetails 2304041000
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id00216007command012queryDetails008flightID0102304041000')
Sending Request
Message Sent!
Response received
Departure Time: Tue, 04/Apr/2023 - 10:0036000000
Airfare: 300.0
Seats Availability: 24
```

When a reservation is made for 1 seat, the seat availability changes from 25 to 24.

```
sashwathravilla@Sashwaths-MacBook-Pro frontend % dfs queryDetails 2304041000
serverIP: 192.168.188.80
marshalled_data: bytearray(b'002id00216007command012queryDetails008flightID0102304041000')
Sending Request
Message Sent!
Response received
Departure Time: Tue, 04/Apr/2023 - 10:0036000000
Airfare: 300.0
Seats Availability: 24
```

When the response for the request to make a reservation has been dropped, the request would have been retransmitted. The request will not be re-executed due to the semantics being at-most-once. The reply from the request that had been previously saved will be retransmitted, therefore producing a correct result.

Additional Features

argparse

Argparse is a Python standard library module that provides a flexible and easy-to-use way to parse command-line arguments in a Python program. It allows developers to define the expected arguments, options, and sub-commands of a command-line interface, and to automatically generate help messages and error handling code.

Testing

Physical (UDP) testing

UDP testing was done in advance to ensure that our server can support clients communicating to it without an initial TCP connection created. This can be shown from our results as one client was able to monitor the flight Id and receives updates when another client has made a booking to that particular flight Id.

Packet Loss

```python
def sendRequest(serverIP: str, serverPort: int, request: bytes):
    retries = 0
    while retries < MAX_RETRIES:
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
            s.settimeout(10)  # set a timeout of 10 seconds
            try:
                s.connect((serverIP, serverPort))
                s.sendall(request)
                print("Message Sent!")
                response = s.recv(MAX_PACKET_SIZE)
                print("Response received")
                s.close()
                break  # exit loop if response is received
            except socket.timeout:
                print(f"Timeout reached, retrying ({retries+1}/{MAX_RETRIES})...")
                retries += 1
                time.sleep(RETRY_DELAY)  # wait before retrying
            finally:
                s.close()
    if retries == MAX_RETRIES:
        print("Max retries reached, giving up.")
        return bytearray()
    else:
        return response
```

As shown by this function, the system is able to handle an example of a packet loss. Since a packet loss will mean that the server does not receive the request sent, or that the client does not receive the response sent, the *sendRequest()* will be able to try 3 times before timeout, hence ensuring that the request made by every client has a response.

Coupled with the semantics checking described earlier, the server will be able to send a response back and achieve its intended effect.