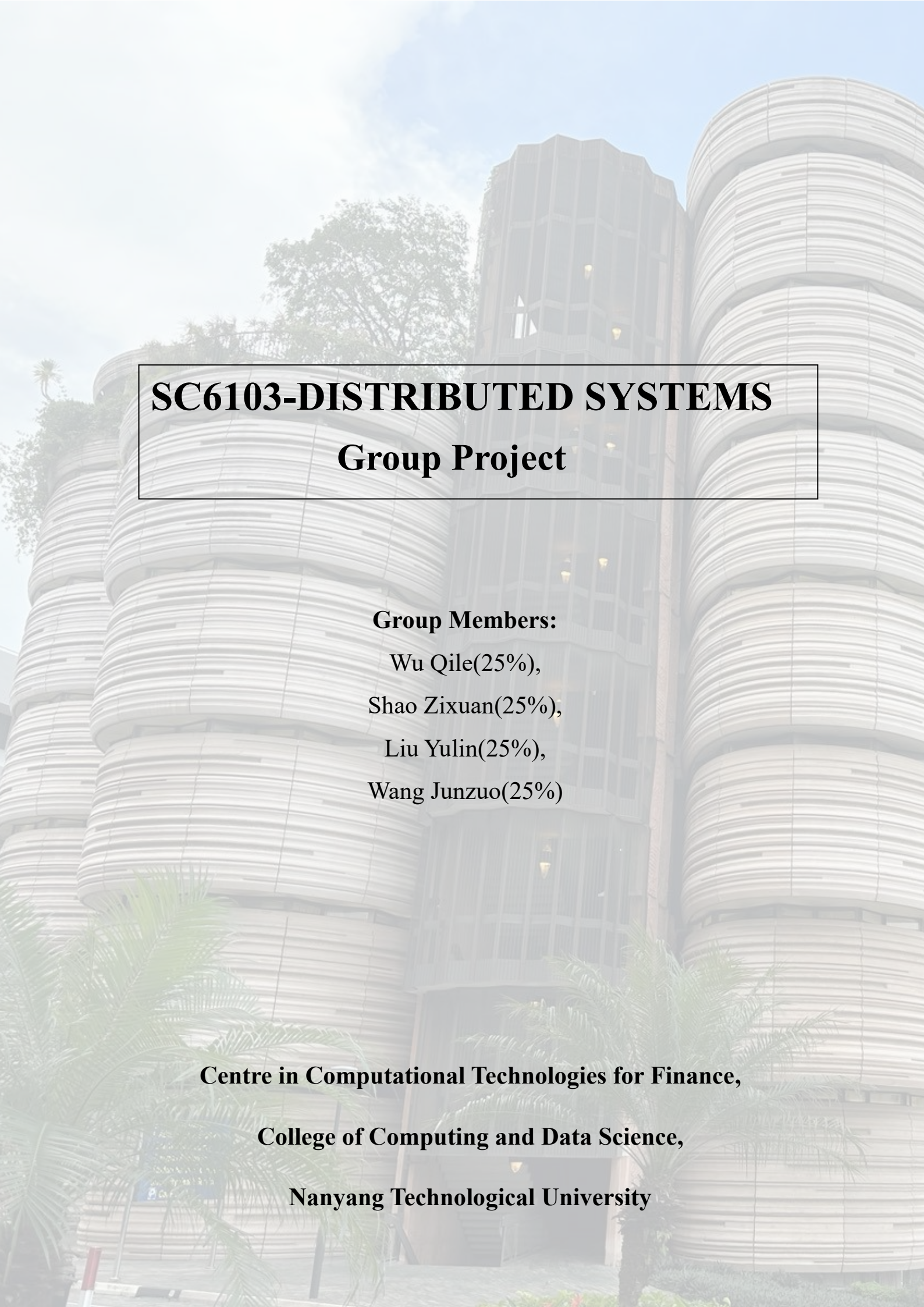

SC6103-DISTRIBUTED SYSTEMS
Group Project

Group Members:

Wu Qile(25%),

Shao Zixuan(25%),

Liu Yulin(25%),

Wang Junzuo(25%)

Centre in Computational Technologies for Finance,

College of Computing and Data Science,

Nanyang Technological University

Content

1. Requirements Analysis

1.1 Functional Requirements

(1)Flight Query Service

- **Description**: Users should be able to query flight identifiers by specifying the source and destination places.
- **Input**: Source (string), Destination (string).
- **Output**: List of flight information.

(2) Flight Information Retrieval Service

- **Description**: Users should be able to retrieve departure time, airfare, and seat availability by specifying the flight identifier.
- **Input**: Flight Identifier.
- **Output**: Departure time, airfare, seat availability, or error message.

(3) Seat Reservation Service

- **Description**: Users should be able to reserve seats on a specified flight.
- **Input**: Flight Identifier, Number of Seats to Reserve.
- **Output**: Confirmation of successful reservation or error message.

(4) Seat Availability Monitoring Service

- **Description**: Users can register to monitor the seat availability of a specific flight and receive updates.
- **Input**: Flight Identifier, Monitoring Interval.
- **Output**: Confirmation of monitoring registration and seat availability updates.

(5) Additional Idempotent Function: Get booking information

- **Description:** Users can retrieve details of their previous bookings, including information about the flight, the number of reserved seats, and the booking status. This feature allows users to check their past reservations and manage their bookings effectively.
- **Input:** None (automatically retrieves the user's booking information).
- **Output:** A list of all bookings made by the user, including booking ID, flight identifier, source, destination, and the number of reserved seats. If there are no bookings, an appropriate message is returned indicating that there are no records available.

(6) Additional Non-Idempotent Function: Randomly select seats

- **Description:** Users can randomly select and reserve seats on a specific flight. This feature automatically chooses a random available seat, simplifying the booking process for users who prefer not to manually select a seat. Every time users use this function, they can get different seats, and that is why it is idempotent.
- **Input:** Flight Identifier

- **Output:** Confirmation of the reserved seat along with the booking details, including the booking ID and flight information.

1.2 Non-Functional Requirements

(1) Both client and server programs must be implemented using UDP socket programming without using existing RMI, RPC, CORBA, or Java object serialization facilities.

(2) Two invocation semantics need to be implemented: at-least-once and at-most-once, with experiments designed to compare the results of these semantics.

2. Backend Design and Implementation

2.1 Backend Overview

The backend of the distributed flight information system is designed to manage flight data, handle client requests, and ensure reliable communication using the UDP protocol. The unmarshalling process is implemented in backend, while the marshalling process is implemented in frontend. The backend system facilitates functionalities such as querying flights, making seat reservations, and providing real-time updates on seat availability to clients. This chapter outlines the architectural design, key components, data management strategies, and implementation details.

2.2 System Architecture

(1)Client-Server Model

The architecture follows a **client-server model** where:

- The **server** listens for incoming UDP packets on a specified port.
- **Clients** send requests for flight information, which the server processes and responds to.

(2)Key Components

①Main Class:

- Responsible for starting the server by initializing the ThreadServer class, which listens for incoming requests.

②ThreadServer:

- Uses a DatagramSocket to receive client requests.
- Parses incoming messages and directs them to appropriate handler classes based on the operation requested (e.g., querying flights, making reservations).

③**DatabaseServer**:

- Manages flight data and user sessions.
- Initializes with a predefined list of flights stored in a HashMap.
- Keeps track of users interacting with the system and their booking information.

④**Model Classes**:

- **Flight**: Contains attributes for flight ID, source, destination, departure time, airfare, and seat availability.
- **BookInfo**: Stores details about flight bookings, including booking ID and associated flight information.
- **User**: Represents a user by storing their IP address and port for communication.

⑤**Utility Classes**:

- **Sender**: Facilitates sending responses to clients using UDP.
- **Arguments**: Contains constants for request types and status codes for message handling.
- **RequestSerializer**: Serializes and deserializes requests, ensuring they conform to a specific format.
- **DuplicateFilterMap**: Filters duplicate requests to avoid processing the same request multiple times.
- **TimedHashMap**: Manages time-limited entries for handling user subscriptions and ensuring timely updates.

（3）Request Handling Process

When a request is received:

①The server extracts the client's IP address and port from the packet.

②It parses the request string to identify the requested operation using the first segment of the message.

③Based on the identified operation, the server calls the corresponding handler class to process the request:

- **QueryFlightBySrcAndDesHandler**: Handles queries for flights based on source and destination.
- **MakeReservationByIdHandler**: Manages seat reservations for specified flights.
- Other handlers include those for querying by flight ID and subscribing to updates.

（4）Data Management

- **Flight Data**:
  - The DatabaseServer initializes a HashMap to store flights with identifiers as keys for quick access.
  - Each flight is represented by an instance of the Flight class, which includes details like airfare and availability.
- **User Management**:
  - A HashSet keeps track of active users to manage their sessions and prevent duplicate bookings.
- **Subscription Handling**:
  - The TimedHashMap is used to store user subscriptions to flight updates, ensuring timely notifications while managing expiration.

2.3 Implementation Details

(1) Unmarshalling

The unmarshalling process is implemented in “ds_backend\src\utils\RequestSerializer.java”. Here are some details and explanations.

①**deserialize(String request):** This method is responsible for converting a string request into a HashMap<String, String>, splitting the input string into key-value pairs.

```java
public static HashMap<String, String> deserialize(String request) {
    String ret = isValidMessageFormat(request);
    HashMap<String, String> req = new HashMap<>();
    if(!ret.equals("true")){
        req.put("error",ret);
        return req;
    }
    String[] componet = request.split(",");
    for (String str : componet) {
        String[] tmp = str.split("=");
        req.put(tmp[0], tmp[1]);
    }
    return req;
}
```

②**isValidMessageFormat(String message)**:This method validates the incoming request format by checking for the presence of required fields and ensuring they conform to expected values.

```
public static String isValidMessageFormat(String message) {
    // Split the message into key-value pairs
    String[] pairs = message.split(",");
    Map<String, String> map = new HashMap<>();

    // Populate the map with keys and values
    for (String pair : pairs) {
        String[] keyValue = pair.split("=", 2);
        if (keyValue.length != 2) {
            //System.out.println("Invalid key-value pair: " + pair);
            return ("Invalid key-value pair: " + pair);
        }
        map.put(keyValue[0].trim(), keyValue[1].trim());
    }

    // Validate each field
    if (!map.containsKey("request")) {
        //System.out.println("Missing 'request' field.");
        return ("Missing 'request' field.");
    } else if (!isValidRequest(map.get("request"))) {
        //System.out.println("Invalid 'request' value: " + map.get("request"));
        return ("Invalid 'request' value: " + map.get("request"));
    }

    if (map.containsKey("source") && !isValidNonEmptyString(map.get("source"))) {
        //System.out.println("Invalid 'source' value: " + map.get("source"));
        return ("Invalid 'source' value: " + map.get("source"));
    }
    if (map.containsKey("destination") && !isValidNonEmptyString(map.get("destination"))) {
        //System.out.println("Invalid 'destination' value: " + map.get("destination"));
        return ("Invalid 'destination' value: " + map.get("destination"));
    }

    if (map.containsKey("id") && !isValidPositiveInteger(map.get("id"))) {
        //System.out.println("Invalid 'id' value: " + map.get("id"));
        return ("Invalid 'id' value: " + map.get("id"));
    }

    if (map.containsKey("seats") && !isValidPositiveInteger(map.get("seats"))) {
        //System.out.println("Invalid 'seats' value: " + map.get("seats"));
        return ("Invalid 'seats' value: " + map.get("seats"));
    }

    if (map.containsKey("timeinterval") && !isValidPositiveInteger(map.get("timeinterval"))) {
        //System.out.println("Invalid 'timeinterval' value: " + map.get("timeinterval"));
        return ("Invalid 'timeinterval' value: " + map.get("timeinterval"));
    }

    if (!map.containsKey("semantic")) {
        //System.out.println("Missing 'semantic' field.");
        return ("Missing 'semantic' field.");
    } else if (!"at-least-once".equals(map.get("semantic")) && !"at-most-once".equals(map.get("semantic"))) {
        //System.out.println("Invalid 'semantic' value: " + map.get("semantic"));
        return ("Invalid 'semantic' value: " + map.get("semantic"));
    }

    return "true";
}
```

(2)Fault Tolerance

①The server implements mechanisms to handle duplicate requests with **DuplicateFilterMap** to filter out already processed messages.

②The **TimedHashMap** ensures that expired subscriptions are removed and that users receive timely updates.

3. Frontend Design and Implementation

3.1 Frontend Overview

The client interacts with the backend server using UDP protocol. Requests are marshalled into strings and sent as UDP packets to the server from the frontend. The server processes these requests and sends back responses, which the client receives and displays.

3.2 Request Handling

The client handles several types of requests, including:

- Querying flights by source and destination.
- Querying flight details by flight ID.
- Making seat reservations.
- Subscribing to flight seat availability updates.
- Randomly selecting a seat on a flight.
- Retrieving booking information.

Each operation is associated with a specific message format sent to the server.

3.3 Invocation Semantics

The system supports two types of **invocation semantics**. The user can choose between these two semantics when performing operations, ensuring that the system behaves according to the desired level of reliability.

(1)At-most-once: Ensures that a message is processed no more than once.

```
const message = `request=QueryFlightBySrcAndDes,source=${source},destination=${destination},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(2)At-least-once: Retransmits a message until an acknowledgment is received, which may result in the operation being processed multiple times (acceptable for idempotent operations like querying flight information).

```
function sendMessage(retransmissions, message) {
  client.send(message, serverPort, serverAddress, (err) => {
    if (err) {
      console.error('Error sending message:', err);
    } else {
      console.log(`Message sent, attempt ${retransmissions}`);
    }

    // Set timeout. If no response is received within the timeout period, resend the message
    timeout = setTimeout(() => {
      if (retransmissions < maxRetransmissions) {
        console.log('No response received, resending message');
        sendMessage(retransmissions + 1, message); // Resend the message
      } else {
        console.log('Max retransmissions exceeded, stopping');
        client.close(); // Close client
        rl.close(); // Close readline interface
      }
    }, timeoutDuration);
  });
}
```

3.4 Marshalling

Different queries have different formats to be marshalled to.

(1)Marshalling for Querying Flights by Source and Destination

```
const message = `request=QueryFlightBySrcAndDes,source=${source},destination=${destination},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(2)Marshalling for Querying Flight by ID

```
const message = `request=QueryFlightById,id=${id},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(3) Marshalling for Reserving Seats

```
const message = `request=MakeReservationById,id=${id},seats=${seats},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(4) Marshalling for Subscribing to Flight Updates

```
const message = `request=SubscribeById,id=${id},timeinterval=${parseInt(timeinterval) * 60000},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(5) Marshalling for Randomly Selecting Seats

```
const message = `request=RandomChooseSeat,id=${id},bookid=${bookId},semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

(6) Marshalling for Getting Booking Information

```
const message = `request=GetBookingInfo,semantic=${semantic}`;
sendMessage(1, Buffer.from(message));
```

4. Demonstration

(1)Query Flights by Source and Destination

```
PS D:\DS6103-main\ds_frontend> node client.js

Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 1
You selected: 1

Enter source: New York
Enter destination: London
Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1

Response received from server:
Query Flights by source and destination:
[identifier=101,sourcePlace=New York,destinationPlace=London,time=1697025600,airfare=500.75,seatAvailability=0]
[identifier=107,sourcePlace=New York,destinationPlace=London,time=1697544000,airfare=480.75,seatAvailability=12]
```

(2)Query flight by ID

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 2
You selected: 2

Enter flight ID: 101
Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1

Response received from server:
Query Flights by identifier:
[identifier=101,sourcePlace=New York,destinationPlace=London,time=1697025600,airfare=500.75,seatAvailability=0]
```

(3)Reserve seats by flight ID

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 3
You selected: 3

Enter flight ID: 107
Enter invocation semantics: 0. At most once 1. At least once 1
Enter number of seats to reserve: 6
Message sent, attempt 1

Response received from server:
Booking Flight Info:
[identifier=107,sourcePlace=New York,destinationPlace=London,time=1697544000,airfare=480.75,seatAvailability=6]
successfully booked 6 seats
```

(4)Subscribe to flight updates: The server will send updates to the client if there are changes to the flight information after each time interval. If there are no updates to the flight information, no information will be sent.

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 4
You selected: 4

Enter flight ID: 107
Enter time interval (minutes): 1
Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1

Response received from server:
user: /127.0.0.1:58714 has successfully subscribed to the flight:
[identifier=107,sourcePlace=New York,destinationPlace=London,time=1697544000,airfare=480.75,seatAvailability=6]
timeout: 60000 milliseconds
```

```
Response received from server:
some one has book this flight:
[identifier=107,sourcePlace=New York,destinationPlace=London,time=1697544000,airfare=480.75,seatAvailability=3]
```

(5)Randomly select seats

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 5
You selected: 5

Enter flight ID: 107
Enter booking ID: 0
Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1

Response received from server:
[bookingId=0, identifier=107, sourcePlace='New York', destinationPlace='London', time=1697544000, airfare=480.75, booked
Time=1729038316462, bookedSeats=3, bookedSeatsId=[7, 9, 5]]
```

(6)Get booking information

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 6
You selected: 6

Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1

Response received from server:
[bookingId=0, identifier=107, sourcePlace='New York', destinationPlace='London', time=1697544000, airfare=480.75, booked
Time=1729038316462, bookedSeats=3, bookedSeatsId=[7, 9, 5]]
```

(7)Exit

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 7
You selected: 7

Thank you for using, the program has exited.
```

(8)An example of at-least-once retry

```
Please select an option:
1. Query flights by source and destination
2. Query flight by ID
3. Reserve seats by flight ID
4. Subscribe to flight updates
5. Randomly select seats
6. Get booking information
7. Exit
Enter your choice (1-7): 5
You selected: 5

Enter flight ID: 107
Enter booking ID: 0
Enter invocation semantics: 0. At most once 1. At least once1
Message sent, attempt 1
No response received, resending message
Message sent, attempt 2
No response received, resending message
Message sent, attempt 3
No response received, resending message
Message sent, attempt 4
No response received, resending message
Message sent, attempt 5
Max retransmissions exceeded, stopping
```

(9) An example of at-most-once Semantic: If the server receives the same request using at-most-once semantic within a certain time interval, it will not process the request again but directly send the same reply message to the client, as shown in the picture.

```
server received message: request=QueryFlightById,id=101,semantic=at-most-once
duplicated request!

Sending response: Query Flights by identifier:
[identifier=101,sourcePlace=New York,destinationPlace=London,time=1697025600,airfare=500.75,seatAvailability=20]
```