Design and Implementation of a Distributed Flight Information System

SC6113 Course Project Report

Instructor: Prof Tang Xueyan

GAO HAN	G2406016G	25% contribution
HAN SHUANGYUE	G2406435C	25% contribution
HUANG ZILIN	G2404841D	25% contribution
YAO FANHUI	G2403762C	25% contribution

October 18, 2024

Abstract

This project develops a distributed flight information system, enabling multiple clients to query, modify, and monitor flight details over a UDP-based client-server architecture. Key functionalities include real-time seat availability tracking, remote method invocation, and resource consistency management. By implementing both at-least-once and at-most-once invocation semantics, we validate and expand upon concepts covered in distributed systems coursework. Our source code is available at the GitHub repository: https://github.com/AvianHan/SC6103_DS

Contents

1 Introduction

1.1 Overview

This project aims to develop a distributed flight information system using UDP-based client-server architecture, exploring key aspects of distributed systems including interprocess communication, resource sharing, and remote method invocation. This system simulates real-world scenarios where multiple clients can concurrently query, modify, and monitor flight information with built-in fault tolerance mechanisms for reliability and scalibility. This report presents a comprehensive view of the system's design, implementation, and experimental evaluation, showcasing the entire development process as per specified requirements.

1.2 Background

1.2.1 Client-Server Architecture

The client-server architecture is a foundational design model in distributed systems, where a server manages resources to provide services while clients request access to those services as shown in Figure 1. In this project, the server holds and updates flight information, responding to multiple clients that may simultaneously query, modify, and monitor flight details. This setup offers centralized control with distributed access, allowing for straightforward implementation and simplified maintenance, as well as providing a flexible framework to efficiently manage concurrent client interactions.

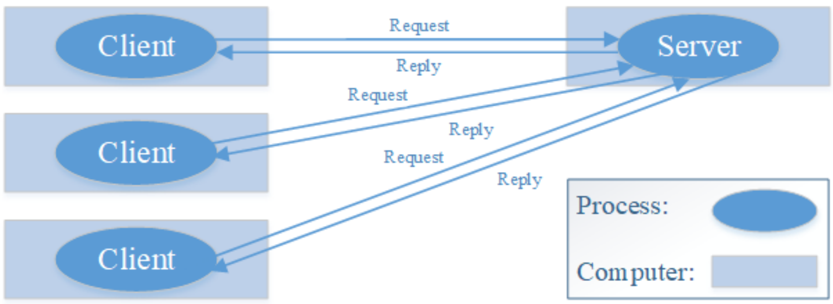


Figure 1: Client-Server Architecture

1.2.2 Interprocess Communication over UDP

UDP (User Datagram Protocol) is a connectionless protocol that supports efficient data transfer without the need for establishing and maintaining a dedicated connection. Although UDP does not inherently guarantee message delivery or order, its simplicity and lower overhead make it ideal for scenarios where speed is prioritized over strict reliability. In this project, UDP facilitates rapid communication between clients and the server, enabling low-latency data exchange crucial for the real-time system. The design compensates for UDP's limitations by implementing fault tolerance mechanisms, ensuring essential data consistency and reliability where necessary.

For a more comprehensive approach, we incorporate reliable request-reply protocols over UDP. This project addresses the unreliability of UDP by using timeout and retry mechanisms for lost messages, as illustrated in Figure 2. When the server's response is delayed or lost, the client resends the request after a timeout to ensure the operation completes. For non-idempotent operations, we employ sequence numbers and a history-based approach to identify duplicate requests and avoid re-execution, thus preserving data consistency. The setup ensures that critical updates are delivered reliably, balancing the benefits of UDP's low latency with necessary fault tolerance to handle message loss and ensure proper system operation.

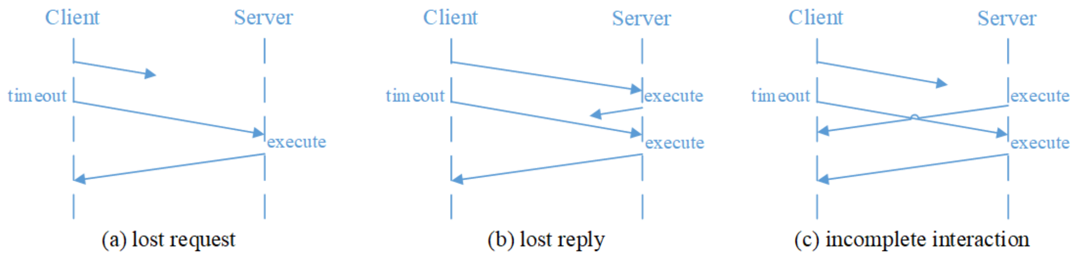


Figure 2: Request-Reply Protocols over UDP

1.2.3 Marshalling and Unmarshalling

Marshalling and unmarshalling are processes crucial to converting structured data into a format suitable for transmission over a network and vice versa. In this project, marshalling is used to serialize complex data structures, such as flight details, into a byte stream that can be sent through UDP sockets. Conversely, unmarshalling is the process of reconstructing this byte stream back into a structured format on the receiving end. These operations are particularly important in distributed systems, where heterogeneous platforms and different data representations necessitate a standardized way of encoding and decoding information for effective communication.

1.2.4 Remote Invocation

Remote Invocation enables a client to execute methods on a remote server as if they were local. In this flight information system, this is simulated over UDP, allowing clients to invoke server-side operations including querying flight information, making reservations, and monitoring seat availability.

Invocation semantics is designed to ensure reliable communication in our UDP-based remote invocation. This project employs two primary invocation semantics, at-least-once and at-most-once, each handling message loss and duplication differently, as summarized in Table 1.

Invocation Semantics	Retransmit Request	Duplicate Filtering	Action
At-least-once	Yes	No	Re-execute method
At-most-once	Yes	Yes	Re-transmit reply

Table 1: Invocation Semantics

- **At-least-once** semantics involves retransmitting requests and re-executing methods when messages are lost, without filtering duplicates. This approach is suitable for idempotent methods, where repeated execution has no adverse effects.

- **At-most-once** semantics ensures each request is executed only once by filtering duplicates and caching responses. For duplicate requests, it retransmits replies instead of re-executing methods, making it ideal for non-idempotent methods that could be compromised by repetition.

These semantics allow the system to meet different reliability needs, balancing operational efficiency with data integrity.

1.3 Methodology

This project adheres to the Software Development Lifecycle (SDLC), leveraging its structured phases to build a reliable and efficient distributed flight information system. Each phase is described in Table 2, along with team roles. The design and implementation draw on foundational concepts from distributed systems and socket programming, as detailed in the course lecture slides by Tang Xueyan (1; 2) and the textbook by Coulouris et al. (3).

SDLC Phase	Team Member(s)
Requirements Analysis and System Design	HAN SHUANGYUE
Server Implementation	GAO HAN, HUANG ZILIN
Client Implementation	HAN SHUANGYUE, YAO FANHUI
Testing	GAO HAN, HUANG ZILIN, YAO FANHUI
Experiments	GAO HAN, HUANG ZILIN, YAO FANHUI
Report Writing	HAN SHUANGYUE

Table 2: Team Roles

2 Requirements Analysis

2.1 Functional Model

The functional model of the system consists of four main services that facilitate interaction between clients and the flight server. Each function is described below in pseudo-code, detailing its operational flow and logic.

Algorithm 1: Query Flight ID

```
Input: source, destination
Retrieve all flight_id where source and destination match the request;
if no matching flights found then
    return Error message: "No flights found";
else if multiple flights found then
    return List of flight_id;
```

Algorithm 2: Query Flight Information

```
Input: flight_id
Retrieve details for the specified flight_id;
if flight_id does not exist then
    return Error message: "Flight not found";
else
    return all the flight information;
```

Algorithm 3: Make Seat Reservation

Input: flight_id, num_seats
Check if `flight_id` exists;
if *`flight_id` does not exist* **then**
  return Error message: "Flight not found";
else if *`num_seats` > `seats_available`* **then**
  return Error message: "Not enough seats available";
else
  Update `seats_available` on database;
  return Acknowledgement message: "Reservation confirmed";

Algorithm 4: Seat Updates and Monitoring

Input: flight_id, monitor_interval
Record the client's Internet address and port number on the server;
Set `monitor_expiration` to current time plus `monitor_interval`;
if *`flight_id` does not exist* **then**
  return Error message: "Flight not found";
Add the client to the list of monitors for the specified `flight_id`;
while *true* **do**
  if *seat reservation is made on `flight_id`* **then**
    Retrieve updated `seats_available`;
    Send updated `seats_available` to registered client(s) via callback;
  for the next seat reservation or monitor;
Remove the client record from the server;
return Acknowledgement message: "Monitor interval expired";

2.2 Data Model

The data model for the flight information system is centered around the `Flight` entity. Each flight entry comprises various attributes that are stored and processed by the system. Table 3 provides a breakdown of the attributes, their sizes, and types.

Attribute	Size	Type
flight_id	4 bytes	int
source	variable	string
destination	variable	string
departure_time	20 bytes	struct
airfare	4 bytes	float
seats_availability	4 bytes	int
baggage_availability	4 bytes	int

Table 3: Data Dictionary

2.3 Behavior Model

The Behavior Model describes the sequence of interactions between the user, client, and server in the flight information system. Figure 3 illustrates the communication flow as follows:

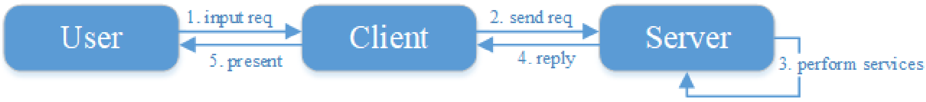


Figure 3: State Transition Diagram

The user inputs a request on the client side, specifying a desired action such as querying flight information or making a reservation. The client sends this request to the server over a UDP connection. Upon receiving the request, the server performs the specified service by querying or updating the relevant information in its database. The server then returns the result of the operation back to the client. Finally, the client presents the result on the user's console, completing the interaction loop.

3 System Design

3.1 Protocol Design

In this system, small-endian and binary encoding are utilized to ensure consistency in data transmission. The data types used for encoding and their respective sizes are as follows:

1. **Integer**: 4 bytes
2. **Float**: 4 bytes
3. **String**: Variable length, prefixed by a 4-byte length descriptor

To facilitate the encoding of structured messages and time data, the system defines specific formats for each, as shown in Tables 4 and 5.

Field	Size	Type
`message_type`	1 byte	—
`payload_len`	4 bytes	int
`payload`	variable	—

Table 4: Message Format

Field	Size	Type
`year`	4 bytes	int
`month`	4 bytes	int
`day`	4 bytes	int
`hour`	4 bytes	int
`minute`	4 bytes	int

Table 5: Time Format

3.2 Module Design

The system is divided into two major components: the Client System and the Server System, as depicted in Figure 4

3.3 Additional Service Design

In addition to the core flight services, the system includes two additional services for managing baggage availability. These services, querying baggage availability and adding baggage, further enhance the system's utility by allowing clients to check and modify the baggage allocation for specific flights. The pseudocode for each service is detailed below.

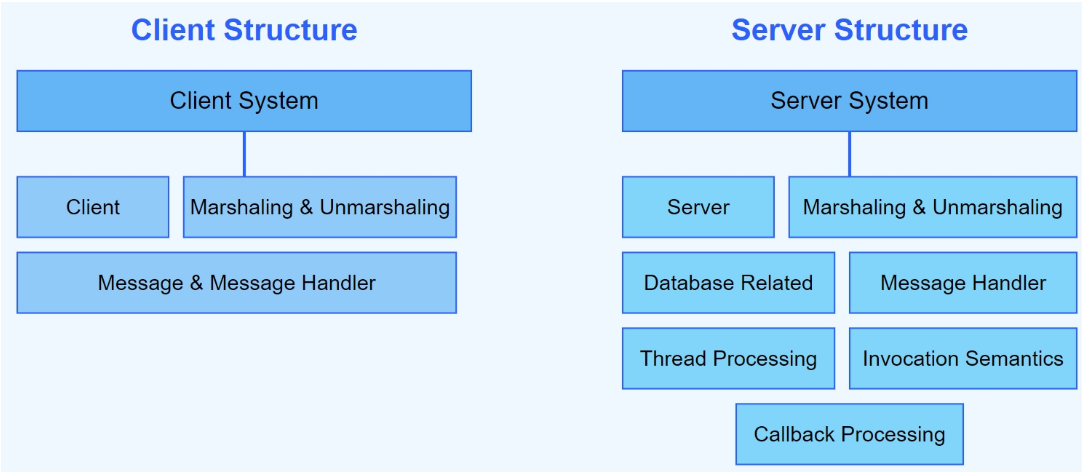


Figure 4: System Structure

Algorithm 5: Query Baggage Availability

Input: `flight_id`
if *`flight_id` does not exist* **then**
  return Error message: "Flight not found";
else
  return `baggage_availability`;

Algorithm 6: Add Baggage

Input: `flight_id`, `num_baggages`
if *reservation is successful* **then**
  return Acknowledgment to client;
  Update `baggage_availability` on server;
else if *`flight_id` does not exist **or** insufficient `baggage_availability`* **then**
  return Error message: "Insufficient baggage availability" or "Flight not found";

4 Detailed Design

4.1 Callback

In this system, the callback functionality enables real-time updates on seat availability to clients. When a client registers for monitoring a flight, the server stores the client's network information (IP and port) and triggers updates whenever a reservation alters the seat count. This mechanism supports asynchronous notifications and enhances the responsiveness of the client-server interaction.

4.2 Resources Consistency

To ensure resource consistency, we utilize a mutex on the database table containing flight information. This synchronization mechanism prevents race conditions when multiple clients simultaneously access or modify flight data. The mutex locks the database during read and write operations, allowing only one operation at a time and preserving the accuracy of seat availability, baggage counts, and other critical data.

4.3 Invocation Semantics

The system implements two invocation semantics, at-least-once and at-most-once, to handle message reliability differently. Both semantics are summarized below, demonstrating how the system achieves consistency and fault tolerance:

- **At-least-once**: If the client doesn't confirm receipt, it re-executes the request.
- **At-most-once**: Stores a history of processed requests, filters duplicates, and directly re-replies with the previous result for any repeated requests.

5 Implementation

5.1 Technology Stack

The development of the distributed flight information system utilized a mix of programming languages, libraries, and tools for efficient data handling, network communication, and concurrency management:

- **Programming Languages:**
  - **C:** Employed for the server-side implementation, focusing on memory efficiency and high-performance data handling.
  - **Java:** Utilized on the client side, enabling cross-platform compatibility and robust GUI components.
- **Database:**
  - **MariaDB:** Used as the primary database for storing and querying flight information, chosen for its reliability and support for SQL operations.
- **Libraries:**
  - **Winsock (Windows) / sys/socket.h (Linux):** Socket libraries to enable UDP-based communication between client and server.
  - **libmariadb:** A library that allows C programs to interface directly with MariaDB for seamless database access and manipulation.
  - **Pthread:** Used for multithreading capabilities on the server, facilitating concurrent client requests.
- **Tools:**
  - **GCC:** The GNU Compiler Collection, used to compile the server code in C.
  - **SQLite Database Browser:** Employed for development and testing database schemas and records.
  - **GDB:** The GNU Debugger, used for debugging server-side operations and tracking low-level memory and network issues.
  - **Wireshark:** Network protocol analyzer utilized to capture and analyze UDP packets, helping ensure accurate data transmission.

5.2 Module Details

The server consists of several modules for handling different aspects of the system as Table 6

Module	Description
`server.c`	Initializes the server, establishes a connection to the database, and manages client request listening.
`data_storage.c / data_connect.c`	Manages interactions with the MariaDB database, including retrieval and updates for flight information.
`flight_service.c`	Contains the core business logic for flight-related operations such as querying, reservations, and baggage management.
`thread_pool.c`	Implements concurrency support, enabling the server to handle multiple client requests simultaneously.
`marshalling.c / unmarshalling.c`	Responsible for data serialization and deserialization, ensuring efficient communication between client and server.
`callback_handler.c`	Responsible for handling the client's callback and notification mechanism.

Table 6: Server Modules Overview

The client consists of several modules for handling different aspects of the system as Table 7

Module	Description
Client.java	Handles client-side operations, including user interaction and sending requests to the server.
Flight.java	Defines the data structure for flights, including attributes like flight ID, source, destination, and other related information.
Marshalling.java	Encodes data into a transmittable format for UDP communication and decodes incoming data from the server.
Message.java	Constructs the message format used for client-server communication, including message types and payload handling.
UserInterface.java	Manages the user interface on the client side, enabling users to input requests and view responses.

Table 7: Client Modules Overview

5.3 Database Structure

The `flights` table stores flight information, with fields such as:

- **flight_id:** Unique identifier for each flight.
- **source_place** and **destination_place:** Origin and destination locations.
- **departure_time:** Timestamp with year, month, day, hour, and minute fields.
- **airfare, seat_availability, baggage_availability:** Other key attributes.

6 Testing

6.1 Function Tests

Extensive testing was conducted for each module, ensuring:

- **Reliability:** Both at-most-once and at-least-once semantics were tested to verify reliability.

- **Performance:** The server was stress-tested with concurrent client connections.
- **Functionality:** Each feature (query, reserve, cancel, and update) was tested for accuracy.

6.2 Semantics Experiments

In our experiments, both invocation semantics were tested to assess their impact on the system's reliability and correctness, especially in the context of non-idempotent operations. By simulating message loss and delays, the following key observations were made:

- **At-Least-Once Semantics:** This approach successfully retransmitted lost requests, ensuring that clients received responses. However, in the presence of duplicate requests for non-idempotent operations, such as baggage addition, the server re-executed these operations, leading to incorrect data states.
- **At-Most-Once Semantics:** Through duplicate detection and response caching, this semantic handled message loss while preserving data integrity. The server filtered out duplicate requests, maintaining correct results for both idempotent and non-idempotent operations.

The comparison experiments of at-least-once and at-most-once semantics present the result shown in Figure **??**.

The experimental results demonstrate that at-least-once semantics is susceptible to errors in non-idempotent operations due to repeated executions. This is evident in operations like `add_baggage`, where duplicates caused inconsistencies.

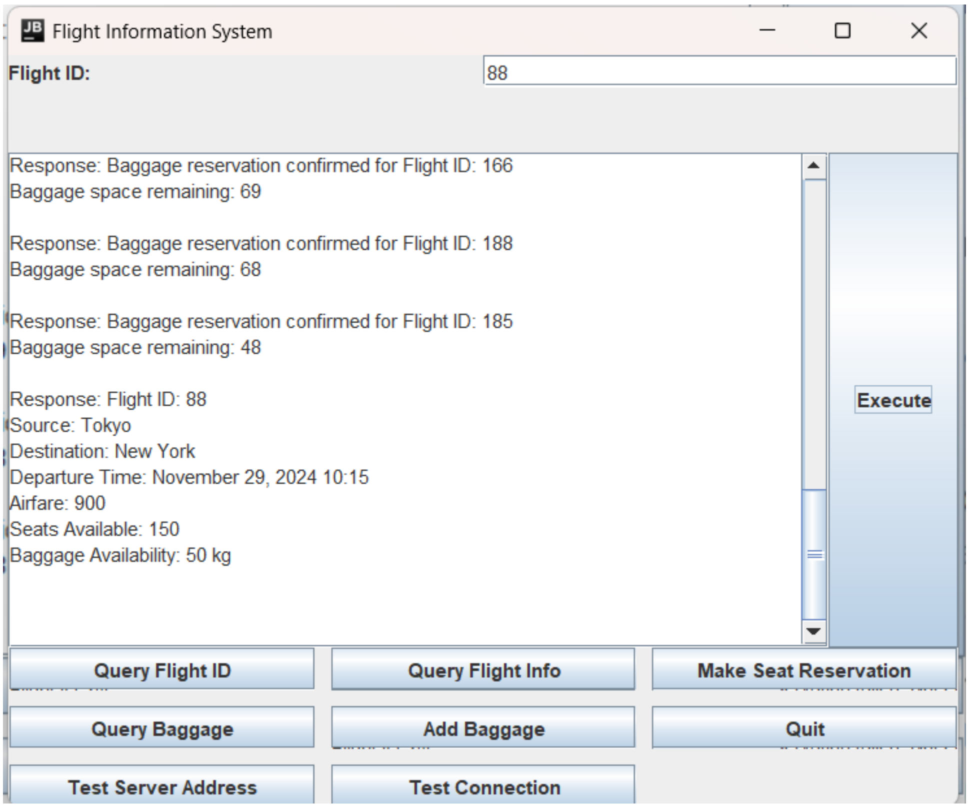
In contrast, at-most-once semantics mitigated these issues by:

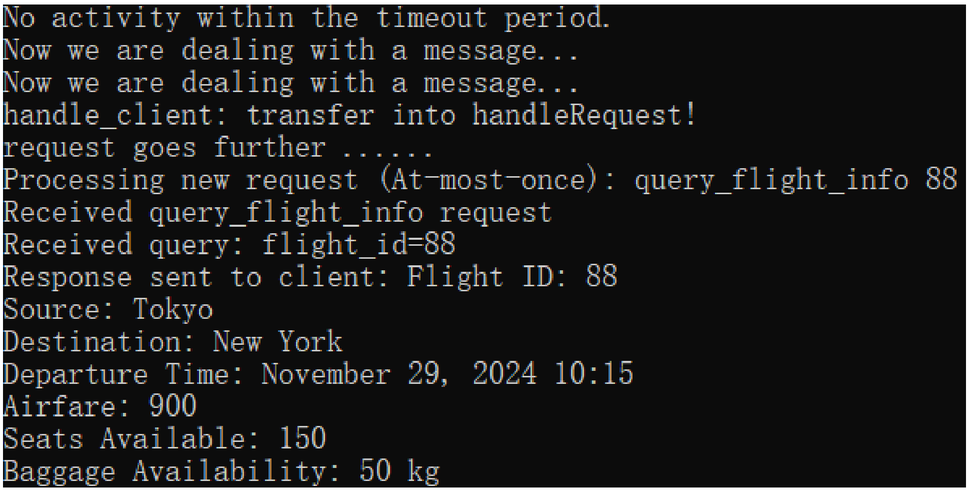
- Implementing request ID tracking and a response cache, which effectively prevented the re-execution of non-idempotent operations.
- Managing reliability through retransmission of responses to duplicate requests, thereby ensuring consistent outcomes regardless of message loss.

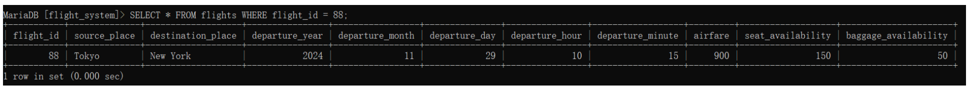
Our findings indicate that while at-least-once semantics may provide a higher level of fault tolerance by re-executing requests, it introduces risks of inconsistency in non-idempotent operations. At-most-once semantics offers a more reliable framework for ensuring accurate and consistent results, particularly where data integrity is crucial. For the overall system, at-most-once invocation semantics is recommended as it guarantees correct outcomes even in adverse conditions, balancing both efficiency and reliability.

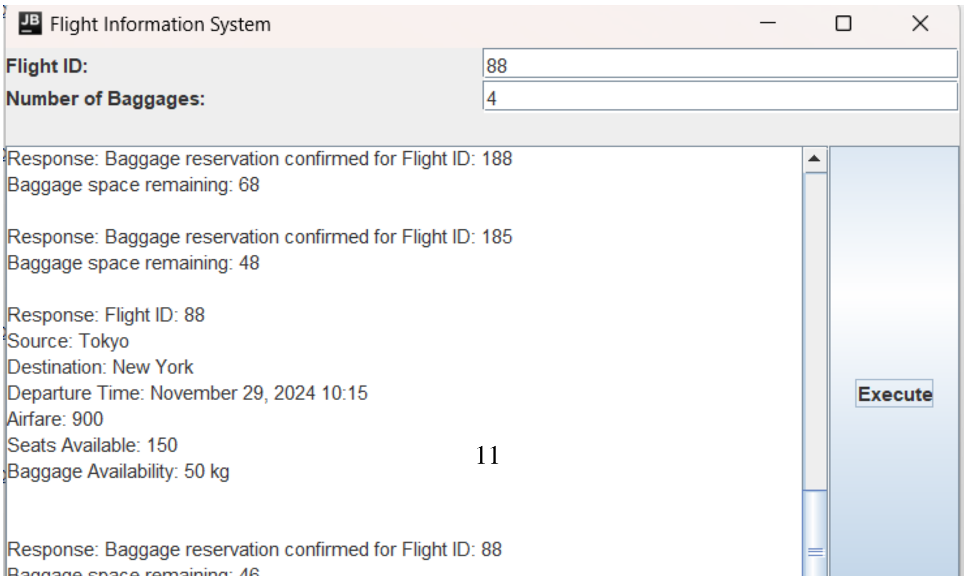
7 Conclusion

The completed system achieves core functionalities such as flight information retrieval, seat reservation, and baggage management with robust real-time updates. Our design incorporates distributed system principles like remote method invocation, concurrency management, and data consistency. Through testing, we confirmed the course-taught invocation semantics: at-least-once semantics handles message loss but risks duplication, while at-most-once semantics preserves data integrity for non-idempotent operations. This project not only meets functional requirements but also validates key distributed system concepts, showcasing the relevance and applicability of theoretical principles in practical implementation.


Flight Information System
Flight ID:
88
Response: Baggage reservation confirmed for Flight ID: 166
Baggage space remaining: 69
Response: Baggage reservation confirmed for Flight ID: 188
Baggage space remaining: 68
Response: Baggage reservation confirmed for Flight ID: 185
Baggage space remaining: 48
Response: Flight ID: 88
Source: Tokyo
Destination: New York
Departure Time: November 29, 2024 10:15
Airfare: 900
Seats Available: 150
Baggage Availability: 50 kg
Execute
Query Flight ID
Query Flight Info
Make Seat Reservation
Query Baggage
Add Baggage
Quit
Test Server Address
Test Connection


No activity within the timeout period.
Now we are dealing with a message...
Now we are dealing with a message...
handle_client: transfer into handleRequest!
request goes further ......
Processing new request (At-most-once): query_flight_info 88
Received query_flight_info request
Received query: flight_id=88
Response sent to client: Flight ID: 88
Source: Tokyo
Destination: New York
Departure Time: November 29, 2024 10:15
Airfare: 900
Seats Available: 150
Baggage Availability: 50 kg


MariaDB [flight_system]> SELECT * FROM flights WHERE flight_id = 88;
| flight_id | source_place | destination_place | departure_year | departure_month | departure_day | departure_hour | departure_minute | airfare | seat_availability | baggage_availability |
| 88 | Tokyo | New York | 2024 | 11 | 29 | 10 | 15 | 900 | 150 | 50 |
1 row in set (0.000 sec)


Flight Information System
Flight ID:
88
Number of Baggages:
4
Response: Baggage reservation confirmed for Flight ID: 188
Baggage space remaining: 68
Response: Baggage reservation confirmed for Flight ID: 185
Baggage space remaining: 48
Response: Flight ID: 88
Source: Tokyo
Destination: New York
Departure Time: November 29, 2024 10:15
Airfare: 900
Seats Available: 150
Baggage Availability: 50 kg
Response: Baggage reservation confirmed for Flight ID: 88
Execute

8 Appendix

8.1 Query Flight ID

The use cases and their corresponding results is shown as Figure 6.

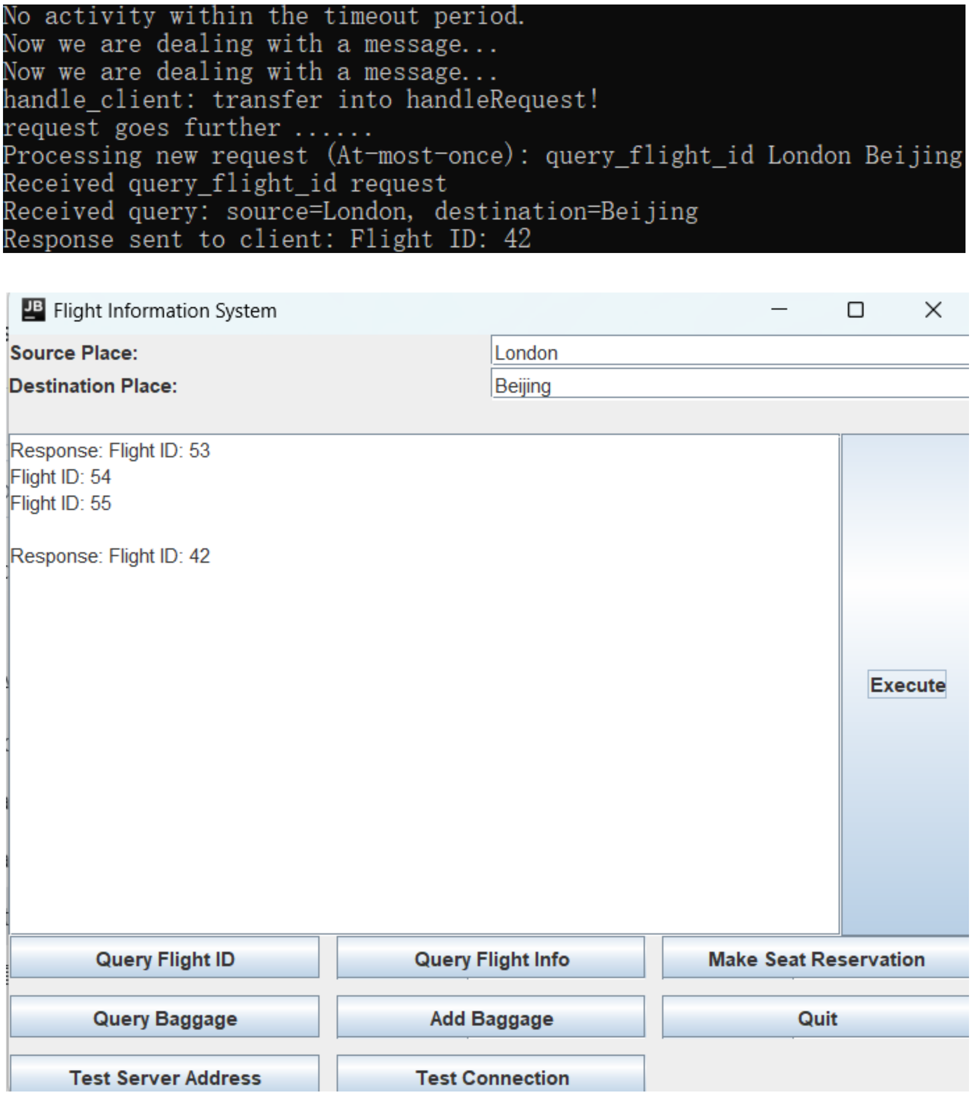


Figure 6: Query Flight ID Function Test Case

8.2 Query Flight Information

The use cases and their corresponding results is shown as Figure 7.

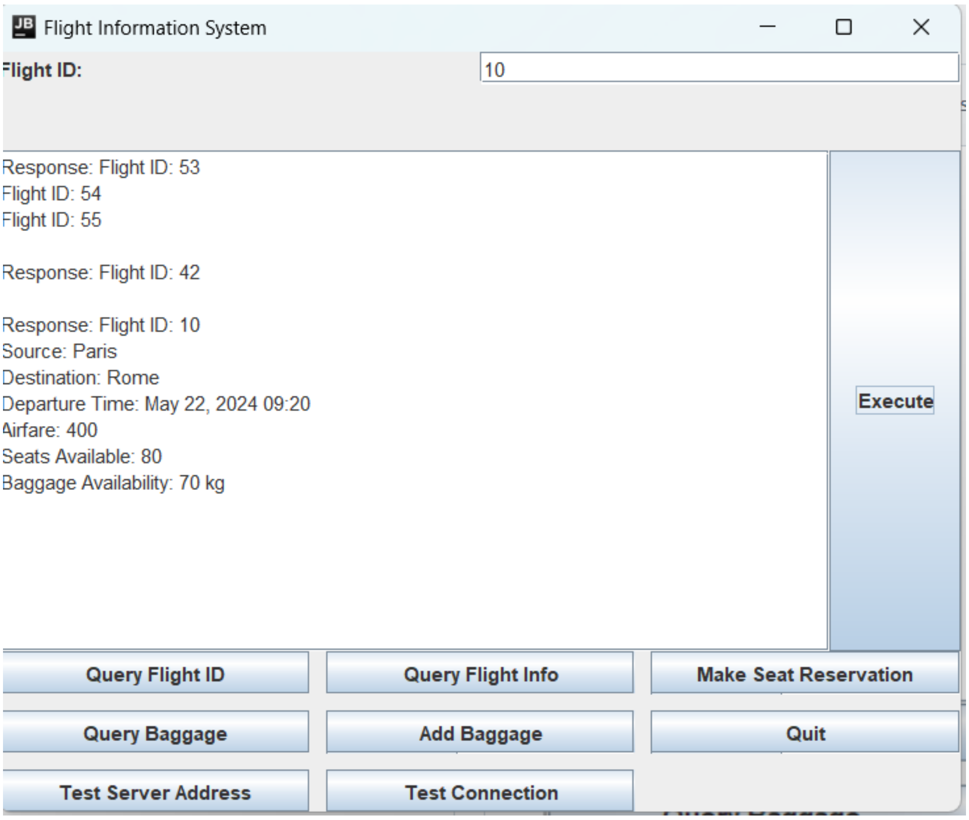
8.2.1 Make Seat Reservation

The use cases and their corresponding results is shown as Figure 8.

8.3 Seat Updates and Monitoring

The use cases and their corresponding results is shown as Figure 9 and **??**.

```
No activity within the timeout period.
Now we are dealing with a message...
Now we are dealing with a message...
handle_client: transfer into handleRequest!
request goes further ......
Processing new request (At-most-once): query_flight_info 10
Received query_flight_info request
Received query: flight_id=10
Response sent to client: Flight ID: 10
Source: Paris
Destination: Rome
Departure Time: May 22, 2024 09:20
Airfare: 400
Seats Available: 80
Baggage Availability: 70 kg
```




```
MariaDB [flight_system]> SELECT * FROM flights WHERE flight_id = 10;
+-----------+--------------+-------------------+----------------+-----------------+---------------+----------------+------------------+---------+-------------------+-----------------------+
| flight_id | source_place | destination_place | departure_year | departure_month | departure_day | departure_hour | departure_minute | airfare | seat_availability | baggage_availability |
+-----------+--------------+-------------------+----------------+-----------------+---------------+----------------+------------------+---------+-------------------+-----------------------+
|        10 | Paris        | Rome              |           2024 |               5 |            22 |              9 |               20 |     400 |                80 |                    70 |
+-----------+--------------+-------------------+----------------+-----------------+---------------+----------------+------------------+---------+-------------------+-----------------------+
1 row in set (0.001 sec)
```

Figure 7: Query Flight Information Function Test Case 1

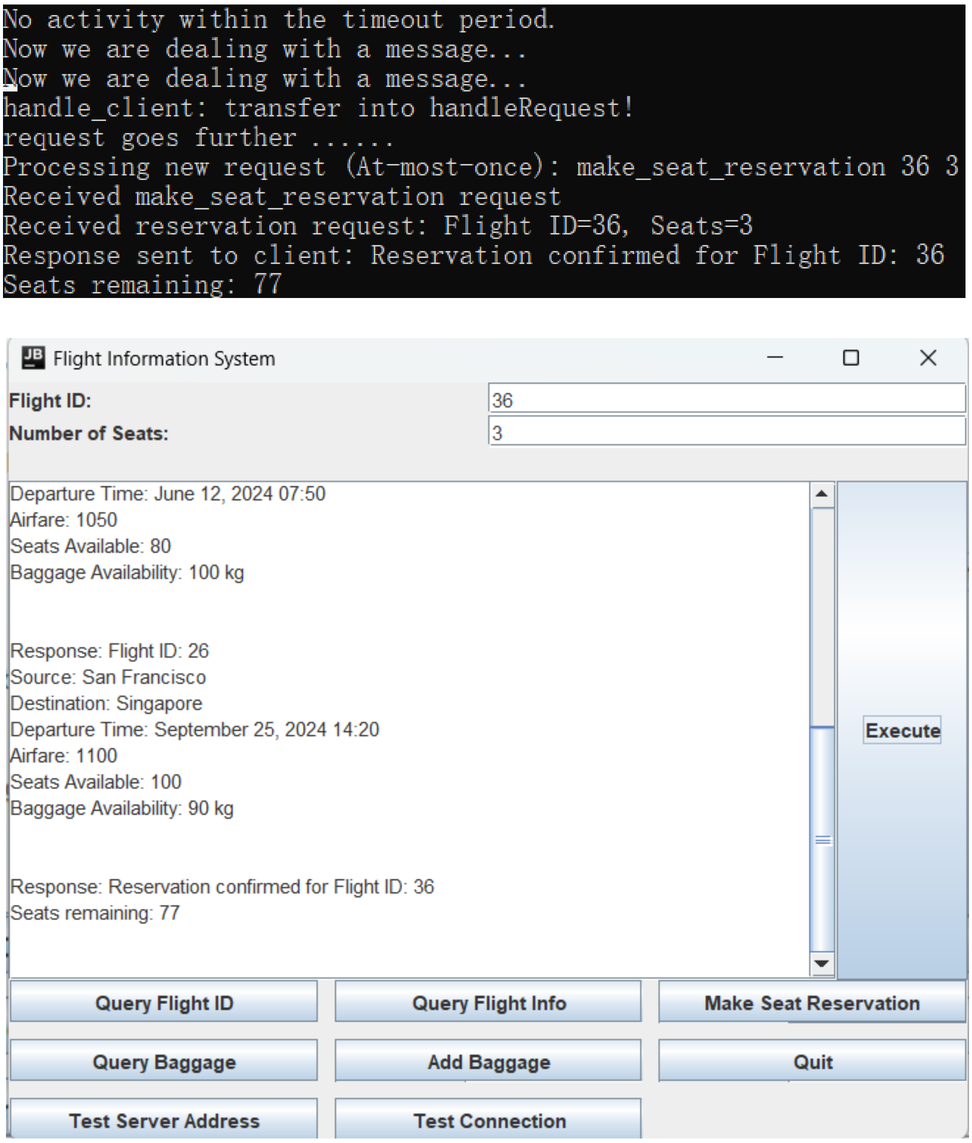
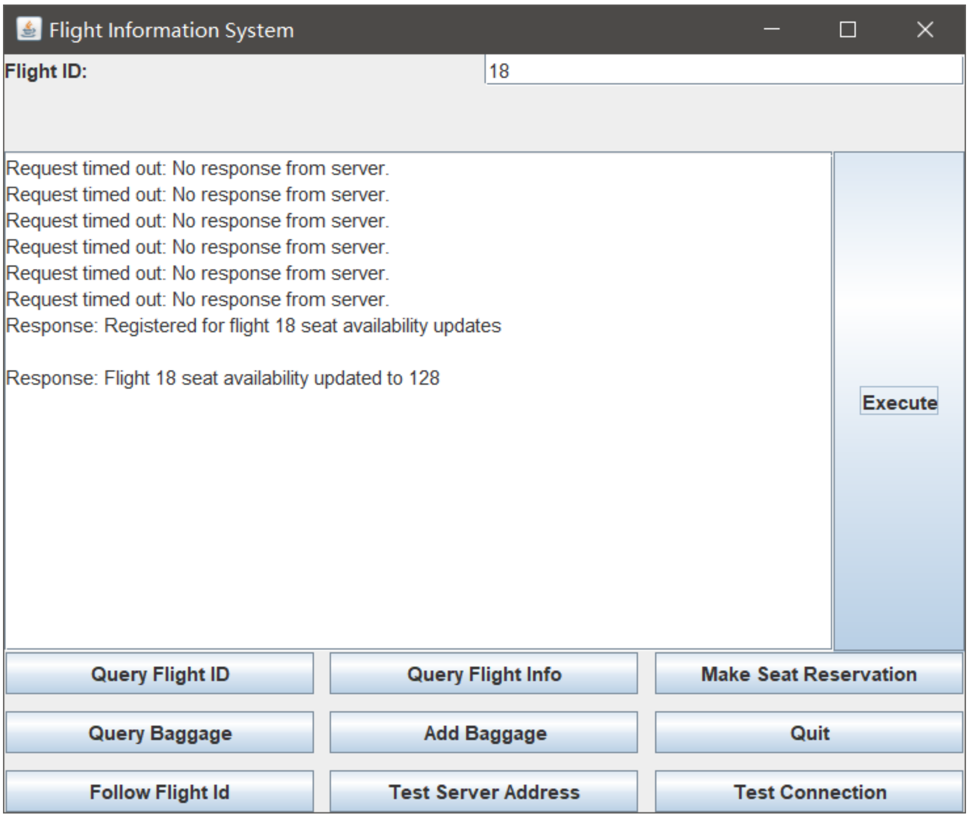


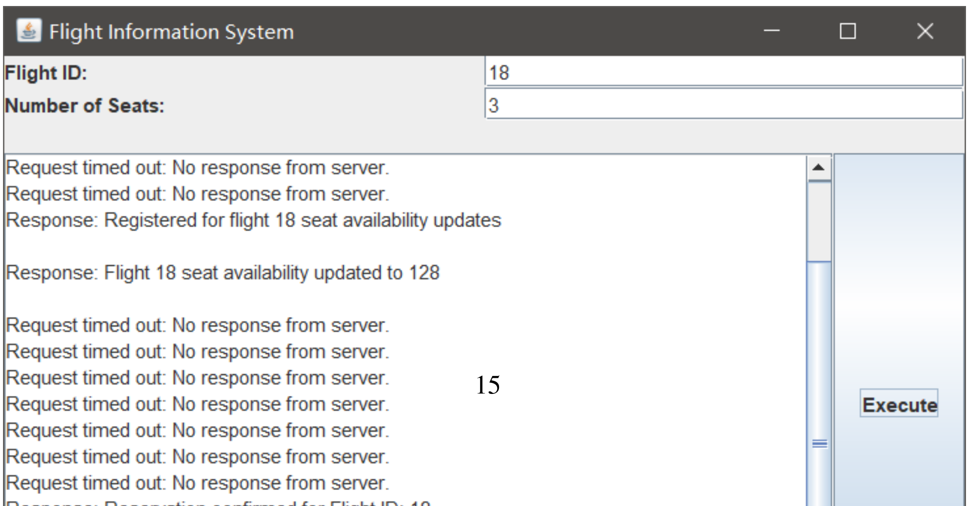
Figure 8: Make Seat Reservation Function Test Case 1

```
F:\ntu\course\6103\SC6103_DS\src\server_c>server.exe at-least-once
Running with at-least-once fault tolerance.
Server is running on port 8080...
already connect to DB!
handle_client: transfer into handleRequest!
Processing new request (At-least-once): 17410864
Received follow_flight_id request
Received follow_flight_id request for flight_id: 18
Response sent to client: Flight monitoring started.

sending reply!
Seats availability changed for flight 18, notifying clients
Successfully sent 43 bytes to the client
```




```
Seats availability changed for flight 18, notifying clients
Successfully sent 43 bytes to the client
handle_client: transfer into handleRequest!
Processing new request (At-least-once): 17410864
Received make_seat_reservation request
Received reservation request: Flight ID=18, Seats=3
Response sent to client: Reservation confirmed for Flight ID: 18
Seats remaining: 125
```



8.4 Query Baggage Availability and Add Baggage

The use cases and their corresponding results is shown as Figure 10.

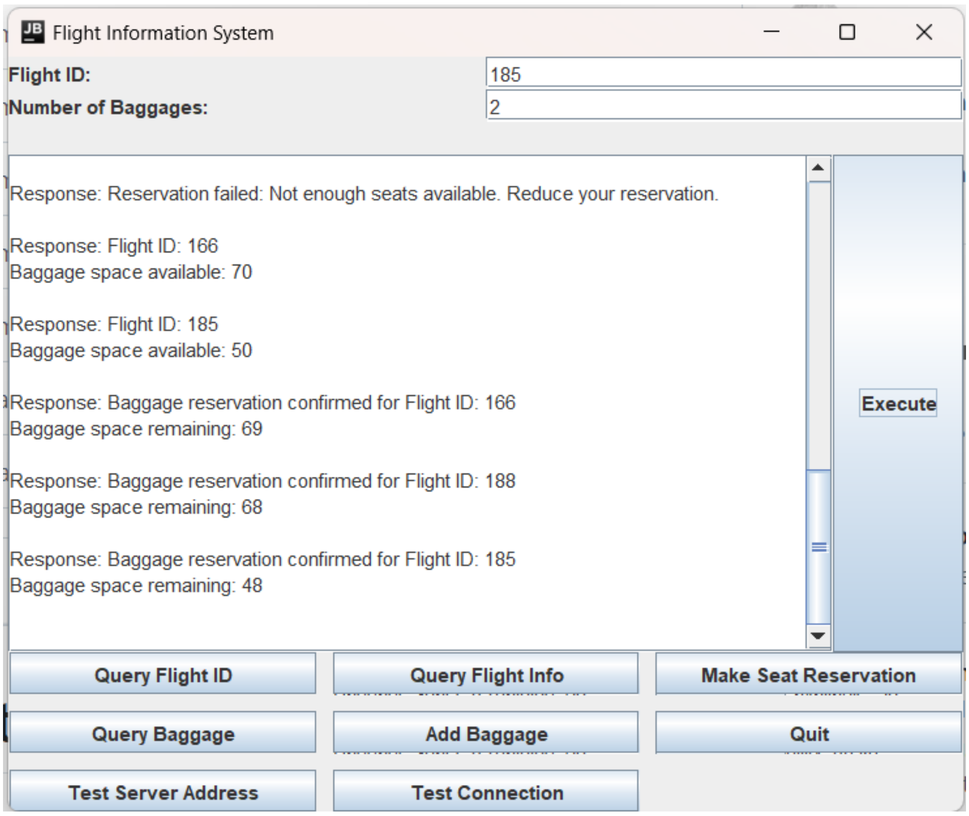
References

[1] Tang Xueyan, Nanyang Technological University, "Lecture Slildes for 2024-2025 Trimester 1 SC6103 Distributed Systems," 2024. Accessed via NTULearn course site.

[2] Tang Xueyan, Nanyang Technological University, "A Tutorial on Socket Programming," 2024. Accessed via NTULearn course site.

[3] T. K. G. B. George Coulouris, Jean Dollimore, "Distributed systems: Concepts and design edition 5," 2012.

```
No activity within the timeout period.
Now we are dealing with a message...
Now we are dealing with a message...
handle_client: transfer into handleRequest!
request goes further ......
Processing new request (At-most-once): query_baggage_availability 185
Received query_baggage_availability request
Received query for baggage availability: Flight ID=185
Response sent to client: Flight ID: 185
Baggage space available: 50
```



MariaDB [flight_system]> SELECT * FROM flights WHERE flight_id = 185;

flight_id	source_place	destination_place	departure_year	departure_month	departure_day	departure_hour	departure_minute	airfare	seat_availability	baggage_availability
185	Tokyo	Shanghai	2025	4	8	21	0	620	100	50

1 row in set (0.000 sec)

```
Now we are dealing with a message...
No activity within the timeout period.
Now we are dealing with a message...
Now we are dealing with a message...
handle_client: transfer into handleRequest!
request goes further ......
Processing new request (At-most-once): add_baggage 185 2
Received add_baggage request
Received baggage reservation request: Flight ID=185, Baggages=2
Response sent to client: Baggage reservation confirmed for Flight ID: 185
Baggage space remaining: 48
```

MariaDB [flight_system]> SELECT * FROM flights WHERE flight_id = 185;

flight_id	source_place	destination_place	departure_year	departure_month	departure_day	departure_hour	departure_minute	airfare	seat_availability	baggage_availability
185	Tokyo	Shanghai	2025	4	8	21	0	620	100	48

1 row in set (0.000 sec)