Distributed Flight Information System - Lab Report

Group Members:
1. Liang Kaiwen - 25% contribution
2. Shi Deming - 25% contribution
3. Lin Hung Chun - 25% contribution
4. Huang Jiajun - 25% contribution

1. Introduction

This report details the design and implementation of a distributed flight information system based on a client-server architecture. The system uses UDP for communication and implements various services including flight querying, seat reservation, and real-time monitoring of seat availability. The project aims to demonstrate practical knowledge of interprocess communication and remote invocation in distributed systems.

2. System Design

2.1 Overall Architecture

The system adopts a client-server architecture and consists of two main components:

1. Server: Stores flight information and processes client requests.

2. **Client**: Provides a user interface for invoking services and displays results.

Communication between the client and server is implemented using UDP sockets, ensuring lightweight and fast data transmission. To enhance reliability, we implemented a custom reliable UDP protocol with timeout retransmission and acknowledgment mechanisms.

2.2 Message Format Design

We designed a flexible message format to accommodate various types of requests and responses. Each message consists of key-value pairs, where each key is represented by an enumerated type `MessageKey`. This design allows for easy extension of message types and efficient marshalling/unmarshalling.

Message structure:

[MessageKey][Type][Length][Value]

- **MessageKey**: An enumerated value identifying the field (1 byte)
- **Type**: Indicates the data type of the value (1 byte)

- **Length**: The length of the value in bytes (1 byte)
- **Value**: The actual data

2.3 Marshalling/Unmarshalling Implementation

We implemented custom marshalling and unmarshalling methods to convert between Java objects and byte arrays for network transmission.

Marshalling process:

1. Convert each field to its byte representation.
2. Prepend each field with its MessageKey, type, and length.
3. Concatenate all fields into a single byte array.

Unmarshalling process:

1. Read the MessageKey, type, and length of each field.
2. Extract the value based on the type and length.
3. Construct a Message object with the extracted values.

2.4 Callback Mechanism for Real-time Updates

We implemented a callback mechanism to support real-time monitoring of seat availability. This feature allows clients to receive updates about seat availability changes without constantly polling the server.

Key components of the callback mechanism:

1. **MonitorCallback class**: Stores information about each monitoring request, including the flight ID, client address, and server socket.
2. **Notification system**: The server maintains a list of active monitoring requests and notifies relevant clients when seat availability changes.
3. **Time-limited monitoring**: Clients can specify a duration for which they want to monitor a flight, after which the callback is automatically removed.

This design enables efficient, real-time updates while minimizing unnecessary network traffic.

2.5 Additional Operations Design

We implemented two additional operations:

1. **Find flight information with the lowest fare by source and destination. (Idempotent)**:

This operation allows users to find the flight with the lowest fare for a given source and destination.

2. **Free Seats (Non-idempotent)**:

This operation allows the freeing of a specified number of seats on a flight, potentially increasing seat availability.

2.6 Reliable UDP Implementation

To address the inherent unreliability of UDP, we implemented a custom reliable UDP protocol. This protocol includes the following key features:

1. **Timeout Retransmission**: If the sender doesn't receive an acknowledgment within a specified timeout period, it retransmits the message.

2. **Acknowledgment Mechanism**: The receiver sends an acknowledgment (ACK) for each successfully received message.

3. **Sequence Numbers**: Each message is assigned a unique sequence number to detect duplicates and ensure ordered delivery.

4. **Sliding Window**: A sliding window protocol is used to manage multiple in-flight packets, improving throughput.

This reliable UDP implementation ensures that messages are delivered even in the presence of network errors or packet loss, while maintaining the performance benefits of UDP.

3. Implementation Details

3.1 Client Implementation

The client provides a console-based interface for users to interact with the system. It handles user input, constructs appropriate request messages, sends them to the server, and displays the responses.

Key features:

- Menu-driven interface
- Request ID generation for duplicate detection
- Timeout and retransmission mechanism
- Acknowledgment handling for reliable message delivery

3.2 Server Implementation

The server listens for incoming requests, processes them, and sends back responses. It maintains the flight database in memory and handles concurrent client connections.

Key features:

- In-memory flight database
- Request processing based on operation type
- Callback mechanism for seat availability monitoring
- Acknowledgment sending for received messages
- Duplicate request detection and handling

Callback Implementation:

1. When a client requests to monitor seat availability, the server creates a `MonitorCallback` object and adds it to a list of active monitors.
2. The server uses a `ScheduledExecutorService` to manage the duration of each monitoring request.
3. When seat availability changes (e.g., due to a reservation or cancellation), the server iterates through the list of active monitors and sends updates to relevant clients.
4. After the specified monitoring duration, the callback is automatically removed from the active list.

This implementation allows for efficient, real-time updates to clients without requiring constant polling, reducing network load and improving responsiveness.

3.3 Invocation Semantics Implementation

We implemented both at-least-once and at-most-once invocation semantics:

At-least-once:

- Client retransmits requests if no acknowledgment is received within a timeout period
- Server processes all requests without checking for duplicates
- Server sends acknowledgments for all received messages

At-most-once:

- Client includes a unique request ID with each request
- Server maintains a history of processed requests
- Server checks for duplicate requests and responds with cached results if found
- Server sends acknowledgments for all received messages, including duplicates

4. Experimental Design and Results Analysis

4.1 Message Loss Simulation

We simulated message loss by randomly discarding a percentage of packets at both client and server sides. This was implemented using a random number generator to decide whether to process or discard each packet.

4.2 Invocation Semantics Comparison Experiment

We designed an experiment to compare the two invocation semantics:

1. Set up the server with a flight having 100 available seats.
2. Use multiple clients to concurrently make seat reservations.
3. Simulate 20% packet loss.
4. Compare the final seat count for both invocation semantics.

4.3 Results Discussion

At-least-once semantics:

- Final seat count: Varied between runs, often less than expected
- Observed duplicate reservations due to retransmissions

At-most-once semantics:

- Final seat count: Consistently correct
- No duplicate reservations observed

The results demonstrate that at-least-once semantics can lead to incorrect results for non-idempotent operations like seat reservation, while at-most-once semantics maintain consistency even in the presence of message loss.

5. Conclusion

This project successfully implemented a distributed flight information system demonstrating key concepts of distributed systems. The use of custom marshalling/unmarshalling, different invocation semantics, callback mechanisms for real-time updates, and fault-tolerance measures provided practical insights into the challenges and solutions in distributed computing.

The implementation of a reliable UDP protocol with timeout retransmission and acknowledgment mechanisms significantly enhanced the system's robustness and reliability. This approach allowed us to maintain the performance benefits of UDP while addressing its inherent unreliability, making our system more suitable for real-world deployment.

The comparison between at-least-once and at-most-once semantics clearly showed the importance of choosing the right invocation semantics for different types of operations, especially in unreliable network conditions. Additionally, the implementation of the callback mechanism for seat availability monitoring demonstrated an efficient approach to providing real-time updates in a distributed system.

Overall, this project provided valuable hands-on experience in designing and implementing reliable distributed systems, balancing performance with consistency and fault tolerance.