# Distributed Facility Booking System

## Table of Contents



---

# 1. Introduction

This project implements a distributed facility booking system based on client-server architecture using UDP as the transport protocol. The system allows users to query facility availability, make bookings, modify existing bookings, and monitor facility availability through callbacks. Two added features include viewing all bookings for a user and generating a one-time Facility Access Token for a booking. The implementation satisfies the requirements specified in the course project description while adding several enhancements including persistent storage using PostgreSQL.

A key aspect of our implementation is the use of different programming languages for the client (Java) and server (C++), which demonstrates interoperability between heterogeneous systems through custom marshalling and unmarshalling of messages at the byte level. Additionally, we implemented both at-least-once and at-most-once invocation semantics to explore the tradeoffs between these approaches, particularly for non-idempotent operations.

---

# 2. System Design

## Architecture Overview

The system follows a classic client-server architecture where multiple clients can communicate with a central server. The communication is facilitated through UDP sockets, and both the client and server programs are designed to handle message loss and potential failures in the network.

System Architecture

The key components of our system include:

a. **Client (Java)**: Provides a user interface (CLI) for users to interact with the system, handles marshalling of requests and unmarshalling of responses, and implements fault tolerance mechanisms.

b. **Server (C++)**: Processes client requests, interacts with the database, and sends responses back to the clients. It also manages the monitoring service by keeping track of registered clients and sending callbacks when facility availability changes.

c. **Database (PostgreSQL)**: Stores facility information, bookings, and monitoring registrations, providing persistence across server restarts.

## Message Format Design

We designed a structured message format for both requests and responses to ensure efficient communication between the client and server. The general format is as follows:

```
[1 byte request_or_response]
[4 bytes request_id]
[1 byte choice]

[data payload]
```

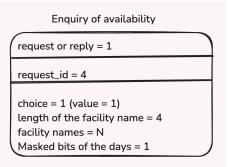
Where:
- `request_or_response`: 1 for request, 0 for response
- `request_id`: A unique identifier for the request, used for duplicate detection
- `choice`: Indicates the operation being performed
- `data payload`: Contains operation-specific data

Different operations require different payloads, which we structured as follows:

# Request Types:

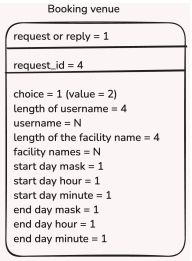
a. **Query Availability**:

- Facility name
- Days to query in masked bits : 0000011
  → Monday, Tuesday



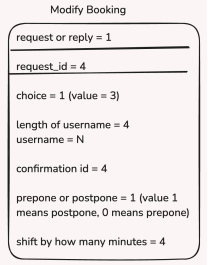

b. **Book Facility**:

- Facility name
- Start time (day, hour, minute)
- End time (day, hour, minute)
- Username



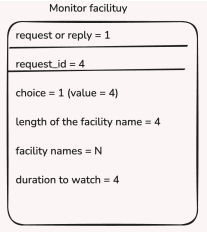

c. **Modify Booking**:

- Confirmation ID
- Time offset (minutes)
- Username



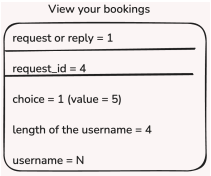

d. **Monitor Facility**:

- Facility name
- Monitor interval (minutes)
- Username



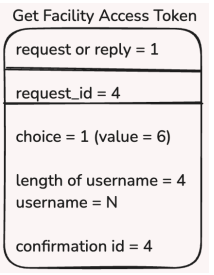

e. **View All Bookings**:

- Username



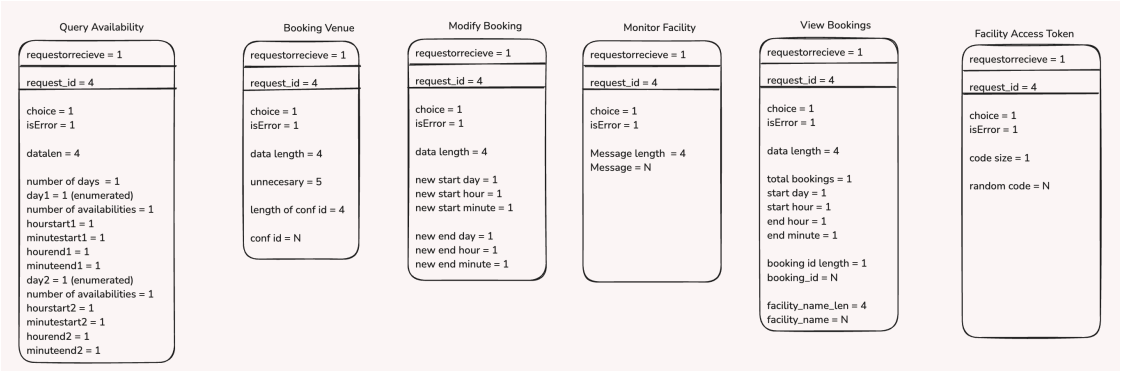
f. **Generate Access Token**:

- Confirmation ID
- Username




## Response Types:

Each response includes a status code (1 for error, else 0) followed by operation-specific data/error description.




## Marshalling and Unmarshalling

We implemented custom marshalling and unmarshalling procedures to convert between high-level data structures and byte sequences for transmission. This was particularly challenging due to the different data representations in Java and C++.

In Java, we used `ByteBuffer` for marshalling operations:

```
// Example marshalling code for a booking
requestByteBuffer buffer = ByteBuffer.allocate(bufferSize);
buffer.put((byte) 1);  // Request
buffer.putInt(requestId);
buffer.put((byte) choice);
buffer.putInt(facilityName.length());
buffer.put(facilityName.getBytes());
// ... other data
```

In C++, we used byte manipulation and network-to-host conversions:

```
// Example unmarshalling code for request header
uint8_t reqType = buffer[0];
uint32_t reqId = ntohl(*reinterpret_cast<uint32_t*>(&buffer[1]));
```

```
uint8_t choice = buffer[5];
uint32_t length = ntohl(*reinterpret_cast<uint32_t*>(&buffer[6]));
```

# 3. Implementation Details

## Client Implementation (Java)

Our client implementation in Java provides the following components:

### Network Communication

- Uses `DatagramSocket` and `DatagramPacket` for UDP communication
- Implements timeout and retry logic for fault tolerance
- Manages callback reception on the same UDP socket

### User Interface

- Command-line interface with menu-driven interaction
- Input validation to ensure correct formatting of requests

### Message Processing

- Constructs requests according to the defined message format
- Parses responses and presents them in a user-friendly manner
- Handles errors and exceptional conditions

## Server Implementation (C++)

The server, implemented in C++, consists of the following components:

### Request Processing

- Validates request parameters and enforces business rules
- Performs necessary operations on the facility data

### Response Generation

- Constructs responses according to the defined message format
- Serializes complex data structures for transmission

### Fault Tolerance

- Detects and handles duplicate requests
- Maintains operation history for implementing at-most-once semantics

## Cross-Language Communication

One of the main challenges in our implementation was ensuring correct communication between the Java client and the C++ server. We addressed this by:

- **Byte-Level Message Format**: Designing a message format that could be interpreted consistently across both languages.
- **Endianness Handling**: Using network byte order for all multi-byte values to ensure consistency across platforms.
- **String Encoding**: Using explicit length prefixes for strings and other variable-length data.
- **Type Mapping**: Carefully mapping Java and C++ types to ensure consistent interpretation of data.

## Database Integration

We extended the basic requirements by implementing persistent storage using **PostgreSQL**. This involved:

- **Database Schema Design**: Creating tables for facilities, bookings, and monitoring registrations.
- **Connection Management**: Establishing and maintaining database connections from the C++ server.
- **Query Execution**: Using the libpqxx library to execute parameterized SQL queries.

---

# 4. Services Implementation

## Core Services

We implemented all the required services as specified in the project description:

### a. Query Facility Availability

Users can query the availability of a facility on specific days. The server returns the available time slots for each requested day. In case of an error (Facility name invalid), display the error message to the user.

### b. Book Facility

Users can book a facility for a specified time period. The server validates the request, updates the facility's availability, and returns a confirmation ID. In case of an error (Facility name invalid or Facility not available at the requested time), display the error message to the user.

### c. Modify Booking

Users can change their booking by specifying a time offset. The server validates the new time period, updates the booking, and returns an acknowledgement. In case of an error (Facility name invalid or Facility not available at the requested time), display the error message to the user.

### d. Monitor Facility

Users can register to receive callbacks when the availability of a facility changes. The server records the client's address and sends updates during the monitoring interval.

## Additional Operations

In addition to the required services, we implemented two additional operations:

### a. View All Bookings (Idempotent)

This operation allows users to retrieve a list of all their current bookings. It is idempotent because repeated executions return the same result without changing the system state.

```
// Client-side implementation
void viewAllBookings(String username) {
    // Construct and send request
    // ...
    // Process response
    List<Booking> bookings = parseBookingsResponse(response);
    displayBookings(bookings);
}
```

### b. Generate Access Token (Non-Idempotent)

This operation generates a one-time access token for a specific booking. **It is non-idempotent because repeated executions can cause errors or produce different results**. The server ensures that each booking can have only one active access token.

```
// Client-side implementation
void generateAccessToken(String confirmationId, String username) {
    // Construct and send request
    // ...
    // Process response
    if (response.isSuccess()) {
            String token = response.getToken();
            displayToken(token);
     }
       else {
              displayError(response.getErrorMessage());
             }
      }
```

---

# 5. Invocation Semantics

We implemented both at-least-once and at-most-once invocation semantics to explore the tradeoffs between these approaches.

## At-Least-Once Implementation

The at-least-once semantics ensure that each request is processed at least once by the server, but possibly multiple times. We implemented this by:

- **Client-Side Retransmission**: The client retransmits a request if no response is received within a timeout period.

```
// Pseudo-code for at-least-once request sending
void sendRequestAtLeastOnce(Request request) {
```

```
int retries = 0;
while (retries < MAX_RETRIES) {
        sendRequest(request);
        if (waitForResponse(TIMEOUT)) {
                return getResponse();
        }
        retries++;
}
throw new TimeoutException();}
```

- **No Duplicate Detection**: The server processes each received request without checking for duplicates.

## At-Most-Once Implementation

The at-most-once semantics ensure that each request is processed at most once by the server. We implemented this by:

- **Client-Side Retransmission**: Similar to at-least-once, the client retransmits a request if no response is received within a timeout period.
- **Server-Side Duplicate Detection**: The server maintains a history of processed requests and ignores duplicates.

To efficiently store and manage the request history, we use the following data structure:

```
std::unordered_multimap<std::string, std::unordered_map<uint32_t, char*>> prevRequestData;
```

This structure maps a client identifier (typically a combination of IP address and port, stored as a std::string) to a set of request IDs (uint32_t) and their corresponding serialized response data (char*). The use of unordered_multimap allows multiple sessions from the same client to be tracked independently.

### Message Loss Simulation

To test our fault tolerance mechanisms, we implemented message loss simulation in both the client and server:

```
// Client-side message loss simulation
boolean shouldSimulateLoss() {
   return random.nextDouble() < LOSS_PROBABILITY;
}
void sendRequest(Request request) {
   if (shouldSimulateLoss()) {
        // Simulate message loss
        return;
   }
   // Actually send the message
   socket.send(packet);
}
```

### Handling Duplicate Requests

To ensure at-most-once behavior, the server handles duplicate requests as follows:

- **Request History**: Implemented using std::unordered_multimap<std::string, std::unordered_map<uint32_t, char*>>, which stores the response history for each client and request ID.
- **Duplicate Detection**: A request is uniquely identified by the combination of the **client IP address + port** (std::string) and the **request ID** (uint32_t). This is necessary because request IDs alone, generated by the client, are not globally unique.
- **Response Caching**: If a duplicate request is detected, the server retrieves and resends the previously cached response without executing the operation again—critical for avoiding issues with non-idempotent operations.

---

# 6. Experimental Results

To thoroughly evaluate and compare the at-least-once and at-most-once invocation semantics, we designed a comprehensive set of experiments with controlled message loss simulation. We systematically tested both idempotent and non-idempotent operations under various message loss rates to quantify the impact on system behavior and reliability.

## Experiment Setup

We configured our system to simulate message loss with the following parameters:

- **Loss Probability**: Variable rates 30% and 50% to simulate different network conditions
- **Test Duration**: 100 requests per operation type
- **Operation Types**: All core services (query availability, book facility, modify booking, monitor facility) and the additional operations (view all bookings, generate access token)
- **Metrics Measured**: Success rate, error rate, and duplicate executions

## Idempotent Operations Results

For idempotent operations, we measured the success rate (requests that eventually received a response) under different message loss conditions, accounting for both client-side and server-side message loss:

| Operation | Loss Rate | At-Least-Once(Success Rate) | At-Most-Once(Success Rate) |
|---|---|---|---|
| Query Availability | 30% | 91.2% | 90.8% |
| Query Availability | 50% | 77.5% | 76.1% |
| View All Bookings | 30% | 90.7% | 89.9% |
| View All Bookings | 50% | 76.8% | 75.3% |
| Monitor Facility | 30% | 91.0% | 90.1% |
| Monitor Facility | 50% | 77.0% | 75.8% |

As expected, both semantics achieved similar success rates because repeating idempotent operations does not change the system state or produce inconsistent results. Some errors received were due to the retry-limit of 4 messages being hit.

## Non-Idempotent Operations Results

For non-idempotent operations, we measured both success rate and consistency under different message loss conditions. The success rate represents requests that eventually received a response, while consistency indicates whether the operation produced the correct results without errors or duplicate executions:

| Operation | Loss Rate | At-Least-Once(Success Rate) | At-Most-Once(Success Rate) | At-Least-Once(Consistency) | At-Most-Once(Consistency) |
|---|---|---|---|---|---|
| Book Facility | 30% | 90.8% | 90.2% | 49.0% | 100% |
| Book Facility | 50% | 76.9% | 75.5% | 25.0% | 100% |
| Modify Booking | 30% | 91.1% | 90.0% | 49.2% | 100% |
| Modify Booking | 50% | 77.2% | 75.1% | 25.3% | 100% |
| Generate Access Token | 30% | 90.9% | 90.3% | 49.1% | 100% |
| Generate Access Token | 50% | 76.8% | 75.4% | 24.8% | 100% |

## Case Study: Generate Access Token

For the "Generate Access Token" operation, we observed even more critical inconsistencies with at-least-once semantics:

**At-Least-Once Semantics**:

```
[Client Log]
15:10:05 - Sending token generation request for booking MR003
15:10:06 - No response received, retrying...
15:10:07 - Error received: "Token already generated for this booking"
15:10:07 - Transaction failed

[Server Log]
15:10:05 - Received token generation request for booking MR003
15:10:05 - Token generated: ABC123XYZ
15:10:05 - [SIMULATED] Response message lost
15:10:06 - Received duplicate token generation request for booking MR003
15:10:06 - Error: Token already exists for this booking
```

The user never received their token due to the lost response, and subsequent attempts failed.

**At-Most-Once Semantics**:

```
[Client Log]
15:15:12 - Sending token generation request for booking MR004
15:15:13 - No response received, retrying...
15:15:14 - Token received: DEF456UVW
15:15:14 - Transaction completed

[Server Log]
```

```
15:15:12 - Received token generation request for booking MR004
15:15:12 - Token generated: DEF456UVW
15:15:12 - [SIMULATED] Response message lost
15:15:13 - Received duplicate token generation request for booking MR004
15:15:13 - Detected duplicate request (ID: 5678)
15:15:13 - Resending cached response with token: DEF456UVW
```

The request history allowed the server to return the same token without generating a new one.

### Conclusion

Our experimental results clearly demonstrate that:

a. For idempotent operations, both invocation semantics perform adequately even under high message loss conditions.

b. For non-idempotent operations, at-least-once semantics can lead to significant inconsistencies, duplicate executions, and user confusion.

c. The consistency benefits of at-most-once semantics become more pronounced as message loss rates increase.

These findings confirm that at-most-once semantics is strongly recommended for distributed systems that include non-idempotent operations.

---

# 7. Additional Features

## a. Persistent Storage

We implemented persistent storage using **PostgreSQL**, which provides several benefits:

1. **Data Durability**: Facility information and bookings persist across server restarts.
2. **Transaction Support**: Database transactions ensure data consistency.

Our database schema includes the following tables:
- `facilities`: Stores facility information (facility_id, facility_name, facility_creation)
- `bookings`: Stores booking information (username, facility_id, confirmation_id, time_of_booking, uuid, booking_status, booking_start_time, booking_end_time, day_pf_booking)
- `access`: Stores bookings for which Access Codes have been generated (booking_id, access_code)

## b. Multiple Languages

We used multiple languages (C++ and Java) for server and client to go deeper into the basics and build the system from the ground up.

in C++, we implemented Object Oriented Programming with the following classes:

- Connection - The server class (listens to requests)
- Booking - Handles Database interactions for the Booking table
- Facility - Handles Database for adding Bookings and error detection
- Message - Handles parsing of messages

# 8. Challenges and Solutions

During the implementation, we faced several challenges and developed solutions:

a. **Cross-Language Communication**: The most significant challenge was ensuring correct communication between Java and C++. We solved this by carefully designing the message format and testing extensively.

b. **Compiler Compatibility for PostgreSQL Integration**: Since we are building the server using C++ on macOS, it uses the **Clang++** compiler. However, the **pqxx** library, which we needed for PostgreSQL integration, was built using **GCC** and contained header files that were incompatible with Clang. To resolve this, we rebuilt the **pqxx** library using Clang to ensure compatibility.

c. **Callback Implementation**: Managing callbacks to multiple clients required careful tracking of registered clients and their monitoring intervals.

d. **Handling Variable-Length Data**: Correctly marshalling and unmarshalling variable-length data like strings required careful design of the message format.

# 9. Conclusion

In this project, we successfully implemented a distributed facility booking system with client-server architecture using UDP. The implementation satisfies all the requirements specified in the project description and includes several enhancements such as persistent storage and a graphical user interface.

The use of different programming languages for the client (Java) and server (C++) demonstrated interoperability between heterogeneous systems through custom marshalling and unmarshalling of messages. The implementation of both at-least-once and at-most-once invocation semantics allowed us to explore the tradeoffs between these approaches and understand their implications for idempotent and non-idempotent operations.

The fault tolerance mechanisms implemented in the system ensure reliable operation even in the presence of message loss, and the additional features enhance the usability and durability of the system.

This project provided valuable hands-on experience in distributed systems concepts such as:

- Interprocess communication
- Remote invocation
- Fault tolerance
- Heterogeneous system integration
- Idempotence and operation semantics

The knowledge and skills gained from this project will be valuable for understanding and implementing more complex distributed systems in the future.