Railway Ticket Booking System using Client-Server Arthitecture

Introduction

• In today's digital age, online booking systems have become an essential part of our daily lives. Whether it's booking a flight, a hotel room, or a train ticket, online booking systems provide a convenient and efficient way to plan our travels. In this project, we aim to design and implement a simple train booking system using socket programming in C.

• The system will allow users to book train tickets online, providing a user-friendly interface and efficient communication between the client and server. The goal of this project is to demonstrate the use of socket programming in a real-world application, while also providing a basic understanding of how online booking systems work. By the end of this project, we will have a fully functional train booking system that allows users to book train tickets with ease.

Objective

Design and implement a simple train seat booking system using a client-server architecture, where:

1. The server maintains a list of trains with their available seats and handles client requests to list trains and book seats.

2. The client connects to the server, sends commands to list trains or book seats, and receives responses from the server.

3. The system allows multiple clients to connect to the server simultaneously, and each client can book seats on different trains.

Key Features

• The server maintains a list of trains with their available seats.

• The client can request to list all trains with their available seats.

• The client can book seats on a specific train, and the server updates the available seats accordingly.

• The system handles multiple client connections simultaneously.

• The client can choose to continue booking or exit the system.

Goals:

• Implement a functional client-server architecture for the train seat booking system. • Ensure thread safety and synchronization in the server-side code.

• Provide a user-friendly interface for the client to interact with the server. • Handle multiple client connections and requests efficiently

System Requirment

**Server-side:**

• Operating System: Linux, macOS, or Windows

• Processor: Any modern processor (Intel Core i3, i5, i7, or equivalent)

• Memory: 512 MB or higher

• Storage: 100 MB or higher

• Compiler: GCC or any C compiler supporting POSIX threads

**Client-side:**

• Operating System: Linux, macOS, or Windows

• Processor: Any modern processor (Intel Core i3, i5, i7, or equivalent)

• Memory: 256 MB or higher

• Storage: 100 MB or higher

• Compiler: GCC or any C compiler

**Network:**

• The server and client should be connected to the same network (e.g., local network or VPN) • The server should have a unique IP address or hostname that the client can connect to **Additional Requirements:**

• The client and server should have the same version of the code to ensure compatibility

• The server should be configured to listen on a specific port (e.g., 8080) that the client can connect to

• The client and server should have proper permissions to create and bind sockets, as well as read and write to files or databases if necessary.

Functionality

**Server-side Functionality:**

1. **Initialize Trains**: The server initializes a list of trains with their available seats.

2. **Listen for Connections**: The server listens for incoming connections from clients. 3. **Handle Client Requests**: The server handles client requests to:

• List all trains with their available seats.

• Book seats on a specific train.

4. **Update Train Availability**: The server updates the available seats for each train based on client bookings.

5. **Send Responses**: The server sends responses to clients for each request, including: • List of trains with available seats.

• Booking confirmation or rejection.

6. **Handle Multiple Clients**: The server can handle multiple client connections simultaneously. **Client-side Functionality:**

1. **Connect to Server**: The client connects to the server using a specific IP address and port. 2. **Send Requests**: The client sends requests to the server to:

• List all trains with their available seats.

• Book seats on a specific train.

3. **Receive Responses**: The client receives responses from the server for each request, including: • List of trains with available seats.

• Booking confirmation or rejection.

4. **Display Information**: The client displays the received information to the user, including: • List of trains with available seats.

• Booking confirmation or rejection.

5. **User Input**: The client accepts user input to:

• Select a train to book seats on.

• Enter the number of seats to book.

• Decide whether to continue booking or exit.

**Common Functionality:**

1. **Error Handling**: Both the server and client handle errors and exceptions, such as: • Connection errors.

• Invalid user input.

• Server-side errors.

2. **Thread Safety**: The server ensures thread safety and synchronization to handle multiple client connections simultaneously.

**Use Cases:**

1. **List Trains**: A user wants to see the list of available trains with their available seats. 2. **Book Seats**: A user wants to book seats on a specific train.

3. **Continue Booking**: A user wants to continue booking seats on different trains. 4. **Exit**: A user wants to exit the system.

**User Interface:**

1. **Command-Line Interface**: The client provides a command-line interface for users to interact with the system.

2. **Menu-Driven Interface**: The client provides a menu-driven interface for users to select options and enter input.

Source Code

// server.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <arpa/inet.h>

#define PORT 8080

#define MAX\_CLIENTS 10

#define MAX\_TRAINS 10

#define MAX\_SEATS 120

typedef struct

{

int available\_seats;

char train\_name[50];

} Train;

Train trains[MAX\_TRAINS];

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

void init\_trains()

{

for (int i = 0; i < MAX\_TRAINS; i++)

{

sprintf(trains[i].train\_name, "Train\_%d", i + 1);

trains[i].available\_seats = MAX\_SEATS;

}

}

void list\_trains(int client\_sock)

{

char buffer[1024];

memset(buffer, 0, sizeof(buffer));

for (int i = 0; i < MAX\_TRAINS; i++)

{

sprintf(buffer + strlen(buffer), "Train: %s, Available Seats: %d\n", trains[i].train\_name, trains[i].available\_seats);

}

if (send(client\_sock, buffer, strlen(buffer), 0) < 0)

{

perror("Send failed");

}

else

{

printf("Train data send");

}

}

void book\_seats(int client\_sock, int train\_id, int num\_seats)

{

pthread\_mutex\_lock(&lock);

if (train\_id >= 0 && train\_id < MAX\_TRAINS)

{

if (trains[train\_id].available\_seats >= num\_seats)

{

trains[train\_id].available\_seats -= num\_seats;

char response[256];

sprintf(response, "Booking successful. %d seats booked on %s.\n", num\_seats, trains[train\_id].train\_name);

if (send(client\_sock, response, strlen(response), 0) < 0)

{

perror("Send failed");

}

}

else

{

char response[] = "Not enough seats available.\n";

if (send(client\_sock, response, strlen(response), 0) < 0)

{

perror("Send failed");

}

}

}

else

{

char response[] = "Invalid train ID.\n";

if (send(client\_sock, response, strlen(response), 0) < 0)

{

perror("Send failed");

}

}

pthread\_mutex\_unlock(&lock);

}

void \*handle\_client(void \*arg)

{

int client\_sock = \*(int \*)arg;

free(arg);

char buffer[1024];

int bytes\_read;

while ((bytes\_read = recv(client\_sock, buffer, sizeof(buffer) - 1, 0)) > 0)

{

buffer[bytes\_read] = '\0';

if (strncmp(buffer, "LIST", 4) == 0)

{

list\_trains(client\_sock);

}

else if (strncmp(buffer, "BOOK", 4) == 0)

{

int train\_id, num\_seats;

sscanf(buffer, "BOOK %d %d", &train\_id, &num\_seats);

book\_seats(client\_sock, train\_id - 1, num\_seats);

}

else if (strncmp(buffer, "NO", 2) == 0)

{

char message[256];

sprintf(message, "Goodbye!\n");

if (send(client\_sock, message, strlen(message), 0) < 0)

{

perror("Send failed");

}

break;

}

}

close(client\_sock);

return 0;

}

int main()

{

init\_trains();

int server\_sock, client\_sock;

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_len = sizeof(client\_addr);

if ((server\_sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket creation failed");

return 1;

}

else

{

printf("Socket creation success");

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(PORT);

inet\_pton(AF\_INET, "127.0.0.1", &server\_addr.sin\_addr);

if (bind(server\_sock, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

{

perror("Bind failed");

return 1;

}

if (listen(server\_sock, MAX\_CLIENTS) < 0)

{

perror("Listen failed");

return 1;

}

printf("Server started. Listening for incoming connections...\n");

while (1)

{

if ((client\_sock = accept(server\_sock, (struct sockaddr \*)&client\_addr, &client\_len)) < 0)

{

perror("Accept failed");

continue;

}

printf("Connected to client IP address %s and port %d...\n", inet\_ntoa(client\_addr.sin\_addr), ntohs(client\_addr.sin\_port));

int \*arg = malloc(sizeof(int));

\*arg = client\_sock;

pthread\_t thread;

if (pthread\_create(&thread, 0, handle\_client, arg) < 0)

{

perror("pthread\_create failed");

continue;

}

}

return 0;

}

// client.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define SERVER\_IP "127.0.0.1"

#define BUFFER\_SIZE 1024

void send\_message(int sock, const char \*message)

{

if (send(sock, message, strlen(message), 0) < 0)

{

perror("Send failed");

exit(1);

}

}

int receive\_message(int sock, char \*buffer)

{

memset(buffer, 0, BUFFER\_SIZE);

int bytes\_received = recv(sock, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received < 0)

{

perror("Receive failed");

exit(1);

}

buffer[bytes\_received] = '\0';

return bytes\_received;

}

void print\_prompt(const char \*buffer)

{

char \*prompt = strstr(buffer, "Enter command");

if (prompt)

{

printf("%s", prompt);

fflush(stdout);

}

}

int main()

{

int sock = 0, exit = 0;

struct sockaddr\_in serv\_addr;

char buffer[BUFFER\_SIZE] = {0};

char input[BUFFER\_SIZE];

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("\n Socket creation error \n");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

if (inet\_pton(AF\_INET, SERVER\_IP, &serv\_addr.sin\_addr) <= 0)

{

printf("\nInvalid address/ Address not supported \n");

return -1;

}

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

printf("\nConnection Failed \n");

return -1;

}

printf("Connected to server.\n");

while (exit == 0)

{

printf("Enter command (LIST / BOOK <train\_id> <num\_seats>): ");

if (fgets(input, BUFFER\_SIZE, stdin) == 0)

{

break;

}

input[strcspn(input, "\n")] = 0;

send\_message(sock, input);

if (receive\_message(sock, buffer) > 0)

{

printf("%s", buffer);

print\_prompt(buffer);

}

while (1)

{

printf("Are you willing to book? (YES/NO): ");

if (fgets(input, BUFFER\_SIZE, stdin) == 0)

{

break;

}

input[strcspn(input, "\n")] = 0;

if (strcmp(input, "NO") == 0)

{

exit = 1;

break;

}

else if (strcmp(input, "YES") == 0)

{

break;

}

else

{

printf("Invalid command!\n");

}

}

}

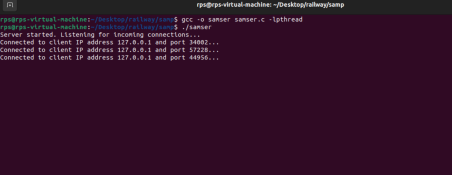
close(sock);

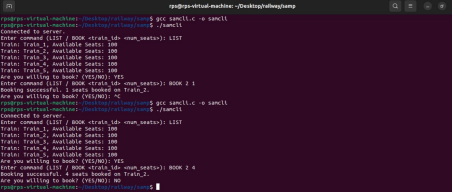
return 0;

}

Output screenshot

Server

Client



Conclusion

The Train Booking System project has successfully developed a functional and user-friendly online platform for booking train tickets. The system provides a convenient, efficient, and cost-effective solution for users. With a robust architecture and user-centric design, the system is poised to become a leading platform for online train ticket bookings, with opportunities for future enhancements and expansions.