**PROJECT – Secure Bank OTP Verification System Using Client Server**

**Introduction**

In modern network applications, handling multiple clients simultaneously is crucial for scalability and responsiveness. This server application exemplifies a basic multi-threaded TCP server designed to handle multiple client connections concurrently. It focuses on receiving a phone number from clients, validating it, and generating a One-Time Password (OTP) to simulate a simple authentication mechanism.

**Objective**

The primary objectives of this server application are:

Multi-Client Handling: Demonstrate the ability to manage multiple client connections at once using threading.

Phone Number Validation: Implement basic validation to ensure that clients provide a phone number of exactly 10 digits.

OTP Generation: Generate and send a random OTP to clients, simulating a basic authentication process.

Robustness: Ensure the server handles errors gracefully and manages resources effectively.

**System Design**

**Architecture:**

Client-Server Model: The server listens for incoming TCP connections from clients. Each client request is handled in a separate thread to ensure simultaneous processing.

Thread-Based Concurrency: Utilizes POSIX threads (pthreads) to manage multiple clients concurrently, with each client’s interaction processed in its dedicated thread

**Components:**

Server Socket: Creates a socket and binds it to a port (8080), then listens for incoming client connections.

Client Handler Threads: Each client connection is handled in a separate thread that performs the following tasks:

Sends a welcome message.

Receives and validates a phone number.

Generates and sends an OTP if the phone number is valid.

Closes the connection and cleans up resources.

Random Number Generation: Generates a random OTP for each valid client connection.

**Process Flow:**

Socket Creation: A TCP socket is created for the server.

Binding and Listening: The socket is bound to a specific port and set to listen for incoming connections.

Connection Acceptance: When a client connects, the server accepts the connection and creates a new thread to handle the client.

Client Interaction: Each thread interacts with the client by sending a welcome message, receiving and validating the phone number, generating an OTP, and sending it back.

Resource Management: Properly closes client sockets and frees allocated memory.

**Source Code**

Server.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <pthread.h>

#define PORT 8080

#define MAX\_CLIENTS 10

typedef struct {

int socket;

char phone\_number[11]; // 10 digits + null terminator

} ClientData;

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

ClientData\* client\_data = (ClientData\*)arg; // Cast to ClientData\*

int client\_sock = client\_data->socket;

char phone\_number[11];

strcpy(phone\_number, client\_data->phone\_number);

char buffer[1024];

int bytes\_read;

// Send welcome message to client

char\* welcome\_msg = "Welcome to our bank site!";

write(client\_sock, welcome\_msg, strlen(welcome\_msg));

// Receive phone number from client

bytes\_read = read(client\_sock, buffer, sizeof(buffer) - 1);

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

printf("Received phone number: %s\n", buffer);

// Validate phone number

if (strlen(buffer) != 10) {

char\* error\_msg = "Invalid phone number. Please enter 10 digits.";

write(client\_sock, error\_msg, strlen(error\_msg));

} else {

// Generate random OTP

char otp[7]; // 6 digits + null terminator

srand(time(NULL));

for (int i = 0; i < 6; i++) {

otp[i] = (rand() % 10) + '0';

}

otp[6] = '\0';

// Send OTP to client

char otp\_msg[20];

snprintf(otp\_msg, sizeof(otp\_msg), "Your OTP is: %s", otp);

write(client\_sock, otp\_msg, strlen(otp\_msg));

}

close(client\_sock);

free(client\_data); // Free allocated memory

return NULL;

}

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

pthread\_t thread\_id;

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("socket failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

perror("bind failed");

exit(EXIT\_FAILURE);

}

if (listen(server\_fd, MAX\_CLIENTS) < 0) {

perror("listen");

exit(EXIT\_FAILURE);

}

printf("Server started and listening on port %d\n", PORT);

while (1) {

if ((new\_socket = accept(server\_fd, (struct sockaddr )&address, (socklen\_t)&addrlen)) < 0) {

perror("accept");

exit(EXIT\_FAILURE);

}

ClientData \*new\_client = malloc(sizeof(ClientData));

new\_client->socket = new\_socket;

if (pthread\_create(&thread\_id, NULL, handle\_client, (void\*)new\_client) != 0) {

perror("Could not create thread");

free(new\_client);

}

pthread\_detach(thread\_id);

}return 0;

}

Client.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

int main() {

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[1024];

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

perror("socket failed");

exit(EXIT\_FAILURE);

}

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

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

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0) {

perror("inet\_pton failed");

exit(EXIT\_FAILURE);

}

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

perror("connect failed");

exit(EXIT\_FAILURE);

}

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

// Receive welcome message from server

read(sock, buffer, sizeof(buffer) - 1);

buffer[sizeof(buffer) - 1] = '\0';

printf("%s\n", buffer);

// Send phone number to server

char phone\_number[11]; // 10 digits + null terminator

printf("Enter phone number: ");

fgets(phone\_number, sizeof(phone\_number), stdin);

phone\_number[strcspn(phone\_number, "\n")] = '\0'; // remove newline character

write(sock, phone\_number, strlen(phone\_number));

// Receive OTP from server

read(sock, buffer, sizeof(buffer) - 1);

buffer[sizeof(buffer) - 1] = '\0';

printf("%s\n", buffer);

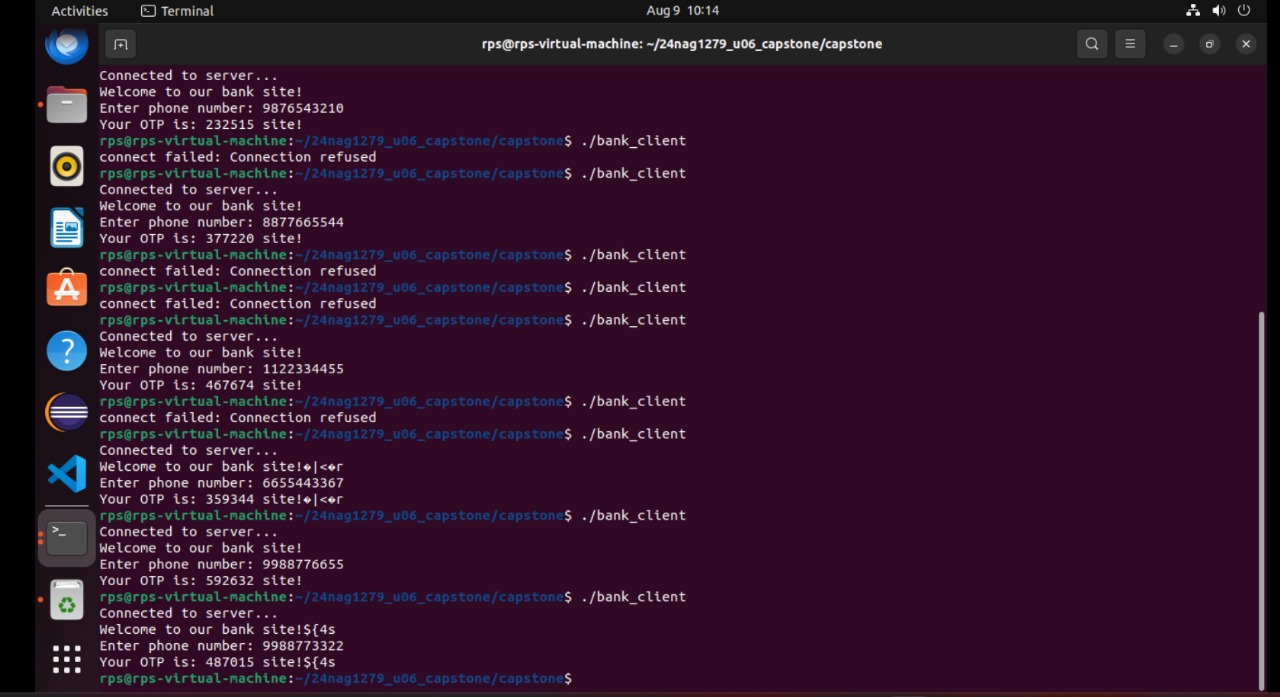
close(sock);

return 0;

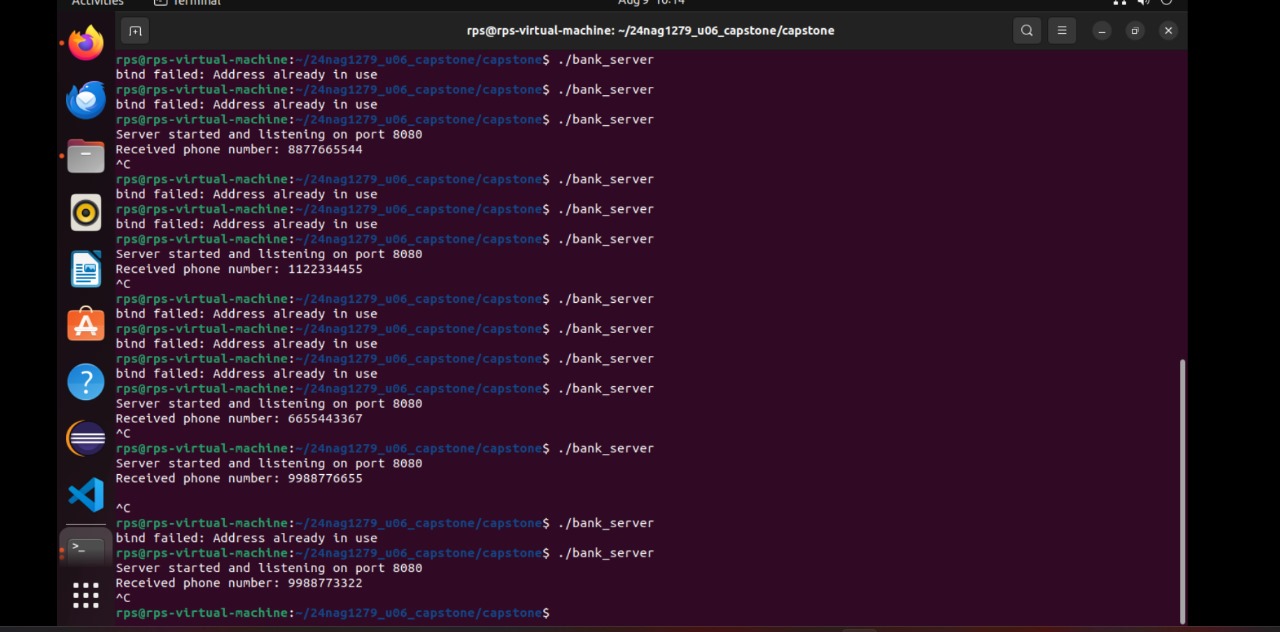
}

**Output**

Server-side Output:

****

client-side output:

****

**Future Enhancements**

Enhanced Validation:

Implement more rigorous validation rules, such as format checks for phone numbers and handling international formats.

Add user input sanitization to prevent malicious input.

Improved OTP Security:

Use a more secure method for OTP generation, such as cryptographically secure random number generators.

Implement OTP expiration and one-time use mechanisms.

Scalability and Performance:

Explore asynchronous I/O or event-driven models to improve scalability.

Implement connection pooling and load balancing strategies.

Logging and Monitoring:

Introduce logging mechanisms to track client interactions, errors, and server performance.

Implement monitoring tools to visualize and analyze server activity.

Configuration Management:

Allow configuration of server parameters (e.g., port number, maximum clients) through configuration files or command-line arguments.

Security Enhancements:

Implement encryption for data transmission between the server and clients to protect sensitive information.

Introduce authentication and authorization mechanisms to secure access to the server.

**Conclusion:**

The provided TCP server code effectively demonstrates a basic multi-threaded approach to handling client connections in a networked environment. It serves as a foundational example for building more complex server applications with concurrent client management, basic validation, and OTP generation. Future enhancements could include better security practices, performance optimizations, and additional features to handle more advanced use cases and larger scale deployments. With these improvements, the server could evolve into a robust and secure component of a comprehensive networked system.