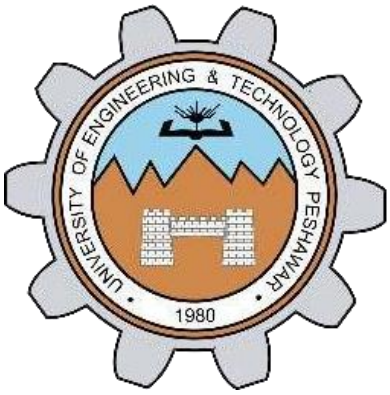
Project Report:

CROSS-OS BRIDGE



Spring 2024

CSE 204L: Operating Systems Lab

Submitted by:

Muhammad Azeem Uddin Khan	22pwcse2167
Muhammad Talha Khan	22pwcse2213
Muhammad Ibad Khan	22pwcse2170

Class Section: **A**

Submitted to:

Engr. Abdullah Hamid

Date:24/06/2024

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar.

Project Report: Server-Client Communication System

1. Problem Statement

We aim to create a server-client communication system where a server can handle multiple client connections. The server will manage tutor and chair resources and respond to client requests. Clients will connect to the server, request a specified number of help sessions and chairs, and then terminate the connection. The server will shut down gracefully upon receiving a termination signal from a client.

2. Algorithm

Server Algorithm[1]

1. Parse command-line arguments to get the number of tutors and chairs.
2. Create a socket and bind it to a specified port[2].
3. Listen for incoming client connections.
4. Accept a connection and create a new thread to handle each client.
5. In each client handler thread:
    - Read data from the client.
    - Check for a termination signal.
    - If a termination signal is received, send an acknowledgment and set a flag to shut down the server.
    - Close the client socket.
6. Shut down the server by user (the server will run in while(1) loop and waits for more clients.

Client Algorithm[1]

1. Parse command-line arguments to get the server IP, number of students, help sessions, and chairs.
2. Initialize Winsock and create a socket[2].
3. Connect to the server.
4. Simulate client actions by sending requests to the server.
5. Send a termination signal to the server.
6. Receive acknowledgment from the server.
7. Close the socket and clean up Winsock resources.

Flowchart

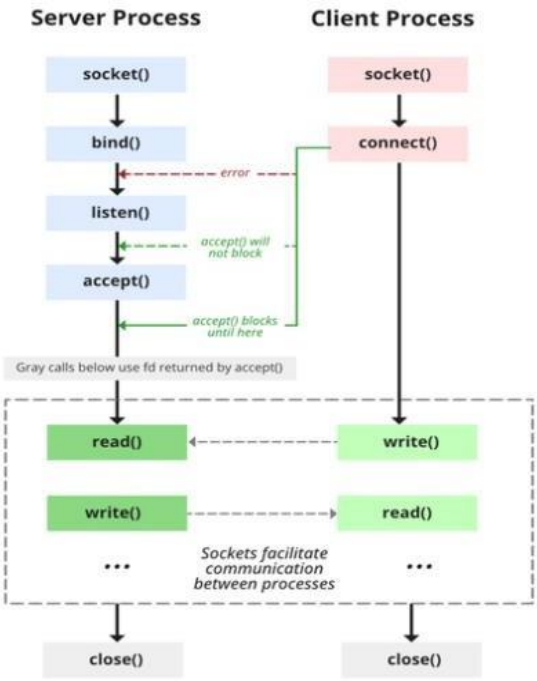


Figure 1:state diagram of server and client model[2]

Functions and Libraries Used

Server Side (server.c) on Linux

- **Functions:**
  - socket(): Creates a socket for communication.
  - bind(): Binds the socket to a specific IP address and port.
  - listen(): Sets the socket to listen for incoming connections.
  - accept(): Accepts an incoming connection request from a client.
  - pthread_create(): Creates a new thread to handle each client connection.
  - pthread_join(): Waits for thread termination.
  - read(), write(), send(), recv(): Functions for reading from and writing to sockets.
  - close(): Closes sockets and cleans up resources.
- **Libraries:**
  - stdio.h, stdlib.h, string.h: Standard C libraries for basic I/O, memory allocation, and string manipulation.
  - unistd.h: Provides access to the POSIX operating system API (used for close() function).
  - arpa/inet.h: Definitions for internet operations (used for struct sockaddr_in).

- pthread.h: POSIX threads library for thread management.

Client Side (client.c for Windows)

- **Functions:**
  - WSAStartup(), WSACleanup(): Initialize and clean up Winsock (Windows Sockets API).
  - socket(), connect(), send(), recv(): Functions for socket creation, connection establishment, and data transmission/reception.
  - sprintf(): Format string into buffer.
  - Sleep(): Pause execution for a specified number of milliseconds.
  - closesocket(): Close the socket and release resources.
- **Libraries:**
  - stdio.h, stdlib.h, string.h: Standard C libraries.
  - winsock2.h: Header for Windows Sockets 2 API.
  - windows.h: Windows API header providing access to core functions like Sleep().
  - time.h: Header providing time-related functions (though not extensively used in this code).

3. Code

Server Code (server.c for Linux)

```c
// server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define SERVER_PORT 8080
#define BUFFER_SIZE 1024

int  tutor_num;
int  chair_num;
int server_socket;

// Function to handle client connections
void *handle_client(void *client_sock);

int main(int argc, char *argv[]) {
    if (argc != 3) {
```

```
        printf("Usage: %s <# of tutors> <# of chairs>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    tutor_num = atoi(argv[1]);
    chair_num = atoi(argv[2]);

    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    int *client_socket;

    // Create socket
    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Setup server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(SERVER_PORT);

    // Bind the socket
    if (bind(server_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("Socket bind failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_socket, 3) < 0) {
        perror("Listen failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    printf("Administration: Listening on port %d\n", SERVER_PORT);

    while (1) {
        // Accept incoming connection
        client_socket = malloc(sizeof(int));
        if ((*client_socket = accept(server_socket, (struct sockaddr
*)&client_addr, &client_len)) < 0) {
            perror("Server: Accept failed");
```

```
            free(client_socket);
            continue;
        }

        printf("Server: Accepted connection from client\n");

        // Create a new thread to handle the client
        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, handle_client, (void
*)client_socket) != 0) {
            perror("Server: Failed to create thread");
            free(client_socket);
            continue;
        }

        // Detach the thread to allow resources to be cleaned up automatically
        pthread_detach(thread_id);
    }

    printf("server: Shutting down.\n");

    // Close the server socket
    close(server_socket);

    return 0;
}

// Function to handle communication with a client
void *handle_client(void *client_sock) {
    int sock = *(int *)client_sock;
    free(client_sock);

    char buffer[BUFFER_SIZE];
    int n;

    while ((n = read(sock, buffer, BUFFER_SIZE)) > 0) {
        buffer[n] = '\0';
        printf("Administration received: %s\n", buffer);

        // Check for termination signal
        if (strcmp(buffer, "TERMINATE") == 0) {
            printf("Administration: Termination signal received. Shutting
down.\n");

            // Send acknowledgment to the client
```

```
            strcpy(buffer, "Termination Acknowledged");
            send(sock, buffer, strlen(buffer), 0);

            // Close the client socket
            close(sock);

            // Exit the loop to shut down the server
            break;
        }
    }

    // Close the client socket if it wasn't already closed due to termination
    if (strcmp(buffer, "TERMINATE") != 0) {
        close(sock);
        printf("Administration: Connection closed with Coordinator.\n");
    }

    return NULL;
}
```

Compilation of server

```
gcc -o server server.c -lpthreads
```

Execution of server

```
./server <number of tutors> <number of chairs>
```

Client Code (client.c for Windows)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>
#include <time.h>

#define SERVER_PORT 8080
#define BUFFER_SIZE 1024

#pragma comment(lib, "ws2_32.lib") // Link with Winsock library
```

```c
void error_exit(const char *message) {
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    if (argc != 5) {
        printf("Usage: %s <server IP> <# of students> <# of help sessions> <# of chairs>\n", argv[0]);
        exit(-1);
    }

    char *server_ip = argv[1];
    int student_num = atoi(argv[2]);
    int help_num = atoi(argv[3]);
    int chair_num = atoi(argv[4]);

    WSADATA wsa;
    SOCKET sock;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
        printf("Failed to initialize Winsock. Error Code: %d\n", WSAGetLastError());
        return 1;
    }

    // Create socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET) {
        printf("Socket creation failed. Error Code: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Setup server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
```

```
        printf("Connection to the server failed. Error Code: %d\n",
WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return 1;
    }

    printf("Coordinator: Connected to server at %s:%d\n", server_ip,
SERVER_PORT);

    // Simulate the client actions
    for (int i = 0; i < student_num; i++) {
        sprintf(buffer, "Student %d needs %d help sessions and %d chairs", i +
1, help_num, chair_num);
        send(sock, buffer, strlen(buffer), 0);
        printf("Coordinator: Sent message: %s\n", buffer);

        // Simulate some delay
        Sleep(1000);
    }

    // Send termination signal
    strcpy(buffer, "TERMINATE");
    send(sock, buffer, strlen(buffer), 0);
    printf("Coordinator: Sent termination signal.\n");

    // Receive acknowledgment from the server
    int n = recv(sock, buffer, BUFFER_SIZE, 0);
    if (n > 0) {
        buffer[n] = '\0';
        printf("Coordinator received: %s\n", buffer);
    }

    // Close the socket
    closesocket(sock);
    WSACleanup();

    return 0;
}
```

Compilation of client[3]

```
gcc -o client client.c -lws2_32
```

Execution of client[3]

```
./client <server IP_adress> <number of students> <no. of helps> <no. of chairs>
```

4. Output (TEST RESULTS)

Server Output

Server: Listening on port 8080

Server received: Student 1 needs 3 help sessions and 2 chairs

Server received: Student 2 needs 3 help sessions and 2 chairs

Server received: TERMINATE

Server: Termination signal received. Shutting down.

Server: Shutting down. //now here the server is terminated because we only have one client but //in normal cases the server will wait for new clients within infinite loop **|while(1)|**

Client Output

Client: Connected to server at 127.0.0.1:8080

Client: Sent message: Student 1 needs 3 help sessions and 2 chairs

Client: Sent message: Student 2 needs 3 help sessions and 2 chairs

Client: Sent termination signal.

Client received: Termination Acknowledged

Practical Uses

1. **Remote Tutoring System:**
    - **Scenario:** The server manages tutor and chair resources, while clients (students) connect to request help sessions and chair availability.
    - **Use Case:** Educational institutions can deploy such systems for virtual tutoring sessions where students can request help from available tutors and reserve chairs for physical meetings.
2. **Resource Management Systems:**

- **Scenario:** The server tracks and manages resources (tutors and chairs) dynamically based on client requests and releases.
- **Use Case:** This system can be adapted for various resource management applications in environments like offices, conferences, or event venues where resources need to be allocated and monitored in real-time.

3. **Distributed Task Processing:**
   - **Scenario:** The server acts as a coordinator that assigns tasks to clients (workers) and gathers results.
   - **Use Case:** In distributed computing environments, tasks can be distributed among multiple clients connected to a central server, allowing parallel processing and efficient task execution.
4. **Messaging and Chat Applications:**
   - **Scenario:** The server manages client connections for messaging and chat applications.
   - **Use Case:** Businesses or social platforms can use similar architectures to enable real-time communication between users across different devices and locations.
5. **IoT (Internet of Things) Device Management:**
   - **Scenario:** The server communicates with IoT devices (clients) to collect data or send commands.
   - **Use Case:** IoT networks often use server-client architectures to manage and control devices remotely, such as in smart homes, industrial automation, or environmental monitoring systems.
6. **Multiplayer Online Games:**
   - **Scenario:** The server manages game sessions and communicates with multiple clients (players) to synchronize gameplay.
   - **Use Case:** Online gaming platforms rely on server-client models to ensure fair gameplay, manage player interactions, and maintain game state across multiple participants.

Applications of the Code

- **Scalability:** The use of threads (pthread.h) in the server allows it to handle multiple client connections concurrently. This scalability is crucial for applications expecting high traffic or requiring simultaneous interactions with multiple clients.
- **Reliability:** The implementation ensures graceful shutdown (TERMINATE signal handling) and proper resource management (socket closure, memory deallocation), which are essential for robust and stable server operations.
- **Cross-Platform Compatibility:** The client-side code (client.c for Windows) demonstrates communication with the server using Winsock API, making it compatible with Windows environments. The server-side code (server.c) utilizes POSIX sockets, ensuring compatibility with Unix-like systems (Linux, macOS).
- **Error Handling:** Both server and client codes incorporate error handling (perror() messages, WSAGetLastError() for Windows errors) to manage unexpected conditions, ensuring reliability and resilience in network communications.

5. Conclusion

This project demonstrates a simple server-client communication system using TCP sockets in C. The server handles multiple clients concurrently using threads and shuts down gracefully upon receiving a termination signal. The client connects to the server, sends requests, and terminates the connection. Proper error handling, resource management, and cleanup ensure the system's reliability and robustness. This setup can be extended to more complex scenarios involving additional client-server interactions and resource management.

REFERENCES

[1] "Simple client/server application in C," GeeksforGeeks. Accessed: Jun. 24, 2024. [Online]. Available: https://www.geeksforgeeks.org/simple-client-server-application-in-c/

[2] "Socket Programming in C," GeeksforGeeks. Accessed: Jun. 24, 2024. [Online]. Available: https://www.geeksforgeeks.org/socket-programming-cc/

[3] sdwheeler, "What is PowerShell? - PowerShell." Accessed: Jun. 24, 2024. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.4