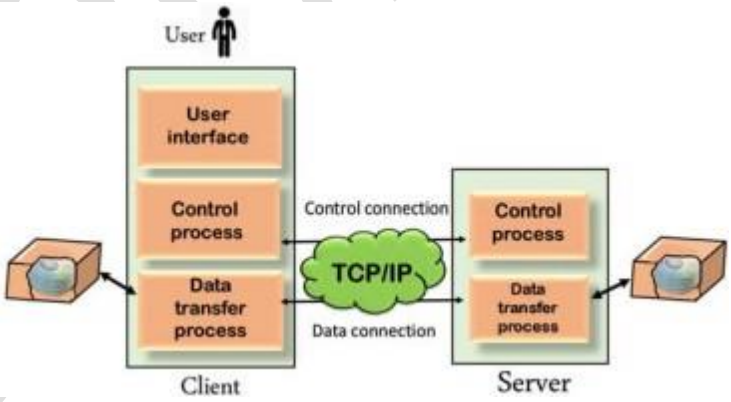
IT304-Computer Networks
Lab 6

Bhavya Shah (202201366)
Ansh Garg (202201343)
Kritarth Joshi (202201338)

FTP is an Internet standard protocol for transferring files between network nodes. RFC-959 defines the FTP protocol specifications. At the very basic level, FTP specifies setting up of two TCP connections – a control connection and a data connection. Control connection is used to pass control information (commands, responses, errors etc.) and the Data connection is used to transfer data.



Exercise

PART-I

1. Implement a TCP Sockets Based File Transfer Protocol

(a) Control and Data Channel Creation

- **Control Channel: This is a TCP connection that handles commands between the client and the server, such as requests for file listings (DIR) and file downloads (GET).**
- **Data Channel: This is a separate TCP connection used exclusively for transferring file data. This separation allows for efficient handling of commands and data transfer.**

Code Implementation:

- **Create processes to handle multiple clients.**

Output:

```
Server is running and waiting for connections...
Client connected.
```

(b) Implementing DIR and GET Commands

- **DIR Command: When a client sends this command, the server responds with a list of files in the specified directory.**
    - **Example request:** `DIR`

Example response:

```
file1.txt
```

```
file2.txt

END_OF_DIR
```

- **GET Command: The client requests a specific file, and the server sends it over the data channel.**

  Example request: `GET file1.txt`

Example

```
OK: Starting file transfer
```

Output:

```
File received successfully: file1.txt

END_OF_FILE
```

(c) Data Transfer Logic

- **The server reads the requested file in chunks and sends it to the client over the data channel.**
- **The client receives the file and saves it locally.**

Expected Output:

```
Time taken to transfer file: 0.42 milliseconds
```

2. Setting Up the Server and Clients

- **Server Setup: Choose one PC to host the server. Place multiple files in a designated directory.**
- **Client Setup: The other two PCs will run multiple client instances, requesting files from the server.**

3. Measuring Transfer Time for a Large File

- **Use a large file to test the transfer speed.**
- **Modify the client code to measure the time before and after the file transfer using `clock()` or similar timing functions.**

Expected Output:

```
Time taken to transfer file: 0.575 milliseconds
```

This indicates how long the transfer took.

---

4. Increasing Number of Clients and Measuring Completion Time

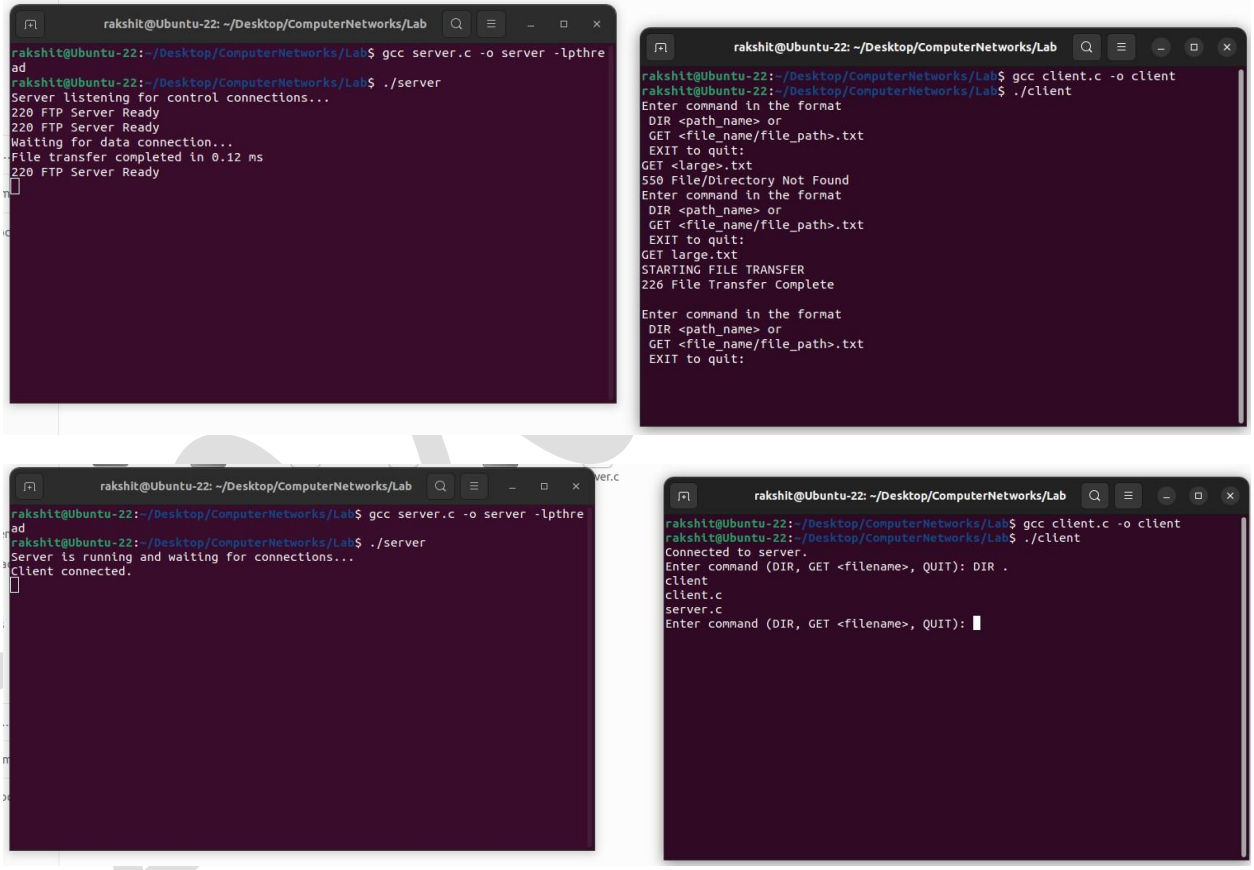
Expected Data Collection: You could structure your data collection like this:

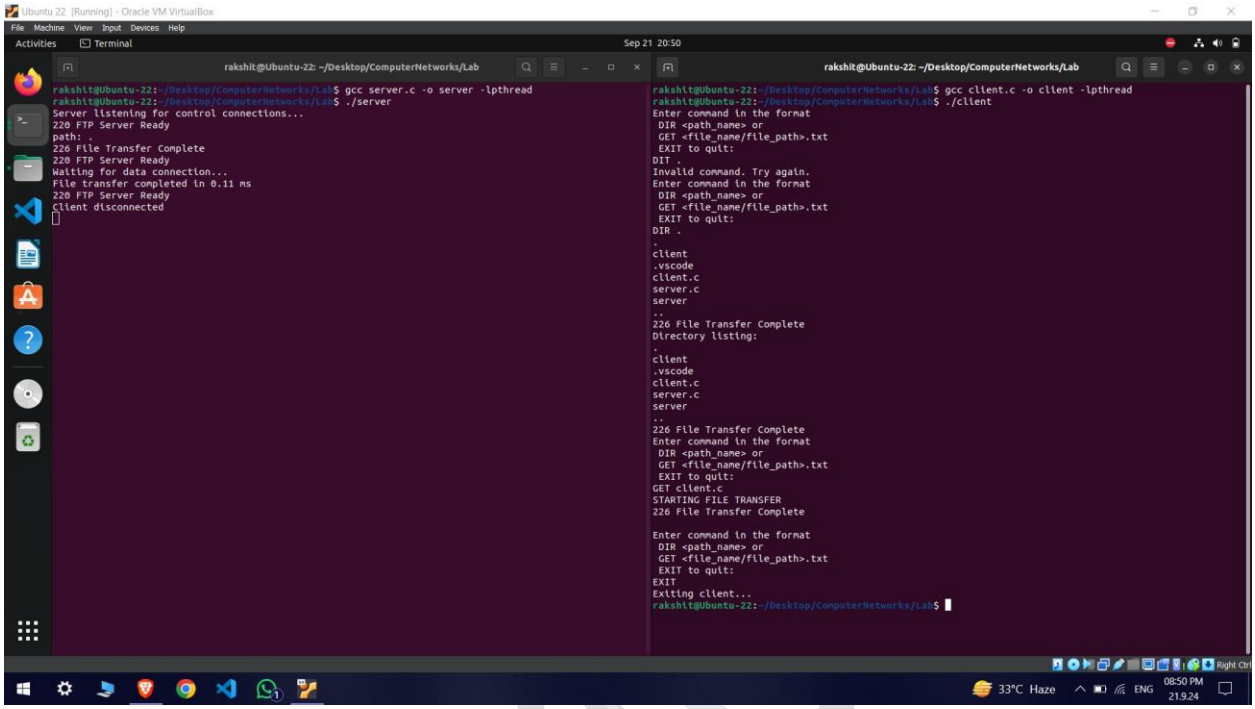
Number of Clients	Completion Time (ms)
1	80
5	100

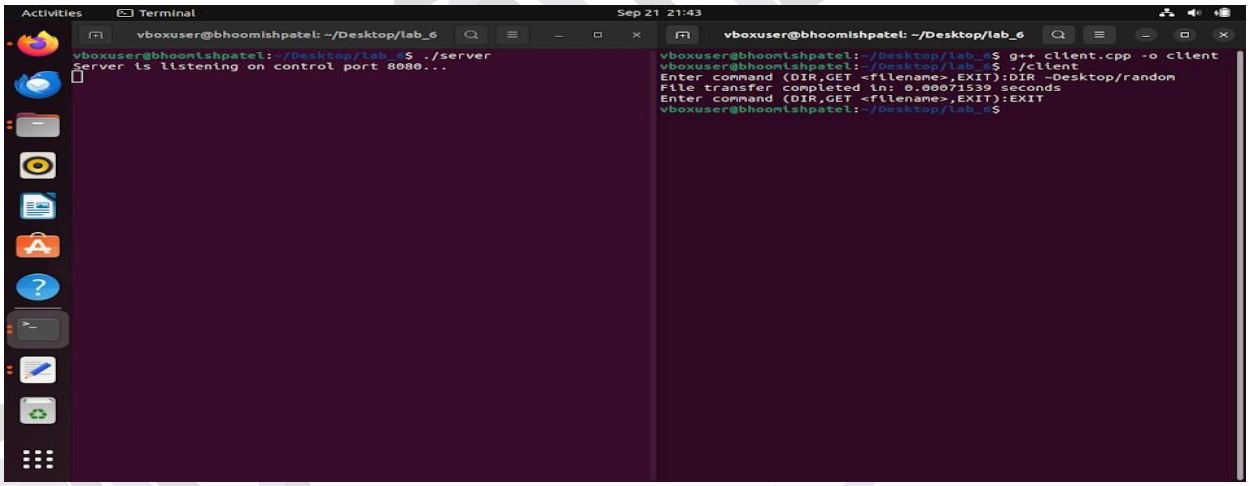
Output: This data allows you to analyze how performance degrades with increasing load.

Conclusion

By completing this assignment, you will gain hands-on experience with TCP sockets, file transfer protocols, and performance analysis in networked applications. The outputs will help you understand the relationship between client load and file transfer efficiency.








```
vboxuser@bhoomishpatel:~/Desktop/lab_6$ g++ client_a.cpp -o client_a
vboxuser@bhoomishpatel:~/Desktop/lab_6$ ./client_a
Enter command (DIR,GET <filename>,EXIT):DIR ~Desktop/random
File transfer completed in: 0.000745 seconds
Enter command (DIR,GET <filename>,EXIT):EXIT
vboxuser@bhoomishpatel:~/Desktop/lab_6$
```

```
vboxuser@bhoomishpatel:~/Desktop/lab_6$ g++ client_b.cpp -o client_b
vboxuser@bhoomishpatel:~/Desktop/lab_6$ ./client_b
Enter command (DIR,GET <filename>,EXIT):DIR ~Desktop/random
File transfer completed in: 0.00111539 seconds
Enter command (DIR,GET <filename>,EXIT):EXIT
vboxuser@bhoomishpatel:~/Desktop/lab_6$
```

Code for question 1-

Server side-

```cpp
// server.cpp

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <cstring>

#include <chrono>

#include <unistd.h>

#include <arpa/inet.h>

#include <dirent.h>

#include <sys/socket.h>

#include <sys/stat.h>

#define CONTROL_PORT 8080

#define DATA_PORT 8081

#define BUFFER_SIZE 1024

const std::string shared_folder = "./shared_files/";
```

```
// Send directory listing

void send_directory(int control_sock) {

DIR *dir;

struct dirent *entry;

char buffer[BUFFER_SIZE];

dir = opendir(shared_folder.c_str());

if (!dir) {

std::string error = "ERROR: Unable to open directory.\n";

send(control_sock, error.c_str(), error.length(), 0);

return;

}

while ((entry = readdir(dir)) != NULL) {

std::string file_name = entry->d_name;

file_name += "\n";

send(control_sock, file_name.c_str(), file_name.length(), 0);

}

closedir(dir);

}

// Send the file to client

void send_file(int data_sock, const std::string &filename) {

std::string filepath = shared_folder + filename;

std::ifstream file(filepath, std::ios::binary);

if (!file) {

std::string error = "ERROR: File not found.\n";
```

```
send(data_sock, error.c_str(), error.length(), 0);

return;

}

char buffer[BUFFER_SIZE];

while (file.read(buffer, sizeof(buffer))) {

send(data_sock, buffer, file.gcount(), 0);

}

if (file.gcount() > 0) {

send(data_sock, buffer, file.gcount(), 0);

}

file.close();

}

// Handle the client request

void handle_client(int control_sock) {

char command[BUFFER_SIZE];

char filename[BUFFER_SIZE];

int data_sock;

struct sockaddr_in data_addr;

socklen_t addr_len = sizeof(data_addr);

data_sock = socket(AF_INET, SOCK_STREAM, 0);

if (data_sock == -1) {

perror("Data socket creation failed");

close(control_sock);

return;
```

```
}

data_addr.sin_family = AF_INET;

data_addr.sin_port = htons(DATA_PORT);

data_addr.sin_addr.s_addr = INADDR_ANY;

if (bind(data_sock, (struct sockaddr *)&data_addr, sizeof(data_addr)) <

0) {

perror("Data bind failed");

close(control_sock);

close(data_sock);

return;

}

if (listen(data_sock, 1) < 0) {

perror("Data listen failed");

close(control_sock);

close(data_sock);

return;

}

while (recv(control_sock, command, BUFFER_SIZE, 0) > 0) {

if (strncmp(command, "DIR", 3) == 0) {

send_directory(control_sock);

} else if (sscanf(command, "GET %s", filename) == 1) {

int data_client_sock = accept(data_sock, (struct sockaddr *)&data_addr,

&addr_len);

if (data_client_sock < 0) {
```

```
perror("Data connection failed");

} else {

send_file(data_client_sock, filename);

close(data_client_sock);

}

} else {

std::string error = "ERROR: Unknown command.\n";

send(control_sock, error.c_str(), error.length(), 0);

}

}

close(control_sock);

close(data_sock);

}

int main() {

int control_sock, client_sock;

struct sockaddr_in control_addr, client_addr;

socklen_t addr_len = sizeof(client_addr);

control_sock = socket(AF_INET, SOCK_STREAM, 0);

if (control_sock == -1) {

perror("Control socket creation failed");

exit(EXIT_FAILURE);

}

control_addr.sin_family = AF_INET;

control_addr.sin_port = htons(CONTROL_PORT);
```

```
control_addr.sin_addr.s_addr = INADDR_ANY;

if (bind(control_sock, (struct sockaddr *)&control_addr,

sizeof(control_addr)) < 0) {

perror("Control bind failed");

close(control_sock);

exit(EXIT_FAILURE);

}

if (listen(control_sock, 1) < 0) {

perror("Control listen failed");

close(control_sock);

exit(EXIT_FAILURE);

}

std::cout << "Server is listening on control port " << CONTROL_PORT <<

"..." << std::endl;

while ((client_sock = accept(control_sock, (struct sockaddr

*)&client_addr, &addr_len)) >= 0) {

handle_client(client_sock);

}

close(control_sock);

return 0;

}
```

Client side-

```
// client.cpp

#include <iostream>

#include <fstream>

#include <chrono>

#include <cstring>

#include <unistd.h>

#include <arpa/inet.h>

#define CONTROL_PORT 8080

#define DATA_PORT 8081

#define BUFFER_SIZE 1024

void receive_file(int data_sock, const std::string& filename) {

std::ofstream file(filename, std::ios::binary);

if (!file) {

std::cerr << "Error opening file" << std::endl;

return;

}

char buffer[BUFFER_SIZE];

int bytes_received;

auto start = std::chrono::high_resolution_clock::now();

while ((bytes_received = recv(data_sock, buffer, BUFFER_SIZE, 0)) > 0) {

file.write(buffer, bytes_received);

}
```

```
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "File transfer completed in: " << elapsed.count() << "

seconds." << std::endl;

file.close();

}

int main() {

    int control_sock, data_sock;

struct sockaddr_in control_addr, data_addr;

char command[BUFFER_SIZE];

char filename[BUFFER_SIZE];

control_sock = socket(AF_INET, SOCK_STREAM, 0);

if (control_sock == -1) {

perror("Control socket creation failed");

exit(EXIT_FAILURE);

}

control_addr.sin_family = AF_INET;

control_addr.sin_port = htons(CONTROL_PORT);

control_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

if (connect(control_sock, (struct sockaddr *)&control_addr,

sizeof(control_addr)) < 0) {

perror("Control connection failed");

close(control_sock);

exit(EXIT_FAILURE);
```

```
}

while (true) {

std::cout << "Enter command (DIR, GET <filename>, EXIT): ";

std::cin.getline(command, BUFFER_SIZE);

if (strncmp(command, "EXIT", 4) == 0) {

break;

}

send(control_sock, command, strlen(command), 0);

if (strncmp(command, "DIR", 3) == 0) {

char buffer[BUFFER_SIZE];

int bytes_received;

while ((bytes_received = recv(control_sock, buffer, BUFFER_SIZE, 0)) >

0) {

buffer[bytes_received] = '\0';

std::cout << buffer;

if (bytes_received < BUFFER_SIZE) break;

}

} else if (sscanf(command, "GET %s", filename) == 1) {

data_sock = socket(AF_INET, SOCK_STREAM, 0);

if (data_sock == -1) {

perror("Data socket creation failed");

continue;

}

data_addr.sin_family = AF_INET;
```

```
data_addr.sin_port = htons(DATA_PORT);

data_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

if (connect(data_sock, (struct sockaddr *)&data_addr,

sizeof(data_addr)) < 0) {

perror("Data connection failed");

close(data_sock);

continue;

}

receive_file(data_sock, filename);

close(data_sock);

} else {

char buffer[BUFFER_SIZE];

int bytes_received = recv(control_sock, buffer, BUFFER_SIZE, 0);

buffer[bytes_received] = '\0';

std::cout << buffer;

}

}

close(control_sock);

return 0;

}
```

Another way of implementing

Code for question 1:->

Server side

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#include <dirent.h>

#include <sys/time.h>


#define PORT 8080

#define DATA_PORT 8081

#define BUFFER_SIZE 1024


char hello_220[256] = "220 FTP Server Ready\n";

char Error_550[256] = "550 File/Directory Not Found\n";

char Complete_226[256] = "226 File Transfer Complete\n";


// Mutex for critical sections

pthread_mutex_t lock;
```

```
// Function declarations
void *handle_client(void *client_socket);
void handle_dir(int control_sock, char* path);
void handle_get(int control_sock, char* filename);
int send_file(int data_sock, FILE* file);

int main()
{
   int server_fd, control_sock;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);

   // Initialize mutex
   pthread_mutex_init(&lock, NULL);

   // Create socket for control channel
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
   {
       perror("Control channel socket failed");
       exit(EXIT_FAILURE);
   }
```

```
    // Bind control socket
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 1);


    printf("Server listening for control connections...\n");


    // Server loop
    while (1)
    {
        control_sock = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);
        if (control_sock < 0)
        {
            perror("Control socket accept failed");
            exit(EXIT_FAILURE);
        }


        // Create a new thread to handle the client
        pthread_t client_thread;
        int *new_sock = malloc(1);
```

```
        *new_sock = control_sock;
        pthread_create(&client_thread, NULL, handle_client, (void*)
new_sock);
        pthread_detach(client_thread);  // Detach thread to allow
independent execution
    }

    close(server_fd);

    // Destroy mutex after the server shuts down
    pthread_mutex_destroy(&lock);
    return 0;
}

void *handle_client(void *client_socket)
{
    int control_sock = *(int*)client_socket;
    free(client_socket);  // Free the memory allocated for socket

    char buffer[BUFFER_SIZE] = {0};

    // Command loop: Process multiple client commands
    while (1)
    {
```

```
memset(buffer, 0, BUFFER_SIZE);

printf("%s", hello_220);

int read_size = read(control_sock, buffer, BUFFER_SIZE);

if (read_size <= 0)

{

    printf("Client disconnected\n");

    break;  // Exit if the client disconnects

}


if (strncmp(buffer, "DIR", 3) == 0)

{

    char path[256];

    sscanf(buffer + 4, "%s", path);

    printf("path: %s\n", path);

    handle_dir(control_sock, path);

}

else if (strncmp(buffer, "GET", 3) == 0)

{

    char filename[BUFFER_SIZE];

    sscanf(buffer + 4, "%s", filename);

    handle_get(control_sock, filename);

}
```

```
        memset(buffer, 0, BUFFER_SIZE);  // Clear buffer for the next
command
    }

    close(control_sock);
    pthread_exit(NULL);
}

void handle_dir(int control_sock, char* path)
{
    DIR *d;
    struct dirent *dir;
    char file_list[BUFFER_SIZE] = {0};

    pthread_mutex_lock(&lock);  // Lock mutex before accessing the
directory

    d = opendir(path);
    if (d)
    {
        while ((dir = readdir(d)) != NULL)
        {
            strcat(file_list, dir->d_name);
            strcat(file_list, "\n");
```

```
        }
        closedir(d);
        send(control_sock, file_list, strlen(file_list), 0);
        send(control_sock, Complete_226, strlen(Complete_226), 0);
        printf("%s", Complete_226);
    }
    else
    {
        send(control_sock, Error_550, strlen(Error_550), 0);
    }

    pthread_mutex_unlock(&lock);  // Unlock mutex after directory handling
}

void handle_get(int control_sock, char* filename)
{
    FILE *file;

    pthread_mutex_lock(&lock);  // Lock mutex for file access

    file = fopen(filename, "rb");  // Open in binary mode
    if (file == NULL)
    {
        send(control_sock, Error_550, strlen(Error_550), 0);
```

```
        pthread_mutex_unlock(&lock);  // Unlock mutex if file not found
        return;
    }

    // Notify client that transfer is starting
    send(control_sock, "STARTING FILE TRANSFER\n", 24, 0);

    // Create the data socket
    int data_sock;
    struct sockaddr_in data_addr;
    int addrlen = sizeof(data_addr);

    data_sock = socket(AF_INET, SOCK_STREAM, 0);
    data_addr.sin_family = AF_INET;
    data_addr.sin_addr.s_addr = INADDR_ANY;
    data_addr.sin_port = htons(DATA_PORT);
    bind(data_sock, (struct sockaddr *)&data_addr, sizeof(data_addr));
    listen(data_sock, 1);

    printf("Waiting for data connection...\n");
    int data_client_sock = accept(data_sock, (struct sockaddr *)&data_addr,
(socklen_t*)&addrlen);

    // Start measuring time for file transfer
```

```
    struct timeval start, end;
    gettimeofday(&start, NULL);

    // File transfer
    int msg;
    msg = send_file(data_client_sock, file);
    if (msg == 0)
    {
        send(control_sock, Complete_226, strlen(Complete_226), 0);
    }
    fclose(file);
    close(data_client_sock);
    close(data_sock);

    // End time measurement
    gettimeofday(&end, NULL);
    double transfer_time = (end.tv_sec - start.tv_sec) * 1000.0;  // Convert to milliseconds
    transfer_time += (end.tv_usec - start.tv_usec) / 1000.0;      // Add microseconds

    printf("File transfer completed in %.2f ms\n", transfer_time);

    pthread_mutex_unlock(&lock);  // Unlock mutex after file handling
```

```
}


int send_file(int data_sock, FILE* file)
{
   char file_buffer[BUFFER_SIZE];
   int bytes_read;


   while ((bytes_read = fread(file_buffer, 1, BUFFER_SIZE, file)) > 0)
   {
       send(data_sock, file_buffer, bytes_read, 0);
   }


   return 0;
}
```

Client Side:->

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


#define CONTROL_PORT 8080
```

```c
#define DATA_PORT 8081
#define BUFFER_SIZE 1024


char hello_220[256] = "220 FTP Server Ready\n";
char Error_550[256] = "550 File/Directory Not Found\n";
char Complete_226[256] = "226 File Transfer Complete\n";



void dir_command(int control_sock,const char* path);
void get_command(int control_sock, const char *filename);


int main()
{
   int control_sock = 0;
   struct sockaddr_in control_addr;
   char input[BUFFER_SIZE];


   // Create control socket
   if ((control_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
       printf("\n Control socket creation error \n");
       return -1;
   }
```

```
    control_addr.sin_family = AF_INET;
    control_addr.sin_port = htons(CONTROL_PORT);


    // Convert IP address to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &control_addr.sin_addr) <= 0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }


    // Connect to server
    if (connect(control_sock, (struct sockaddr *)&control_addr,
sizeof(control_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }


    // Command loop to read input from the terminal
    while (1) {
        char buffer[BUFFER_SIZE] = {0};
        printf("%s",buffer);
        memset(input,0,BUFFER_SIZE);
```

```
        printf("Enter command in the format \n DIR <path_name> or\n GET
<file_name/file_path>.txt \n EXIT to quit: \n");
        scanf("%[^\n]s",input);
        getchar();


        // Parse the input and execute corresponding command
        if (strncmp(input, "DIR", 3) == 0)
        {
            char path[BUFFER_SIZE];
            sscanf(input + 4, "%s", path);
            dir_command(control_sock,path);
        }
        else if (strncmp(input, "GET", 3) == 0)
        {
            char filename[BUFFER_SIZE];
            sscanf(input + 4, "%s", filename); // Extract filename
            get_command(control_sock, filename);
        }
        else if (strncmp(input, "EXIT", 4) == 0)
        {
            printf("Exiting client...\n");
            break;
        }
        else
```

```
        {
            printf("Invalid command. Try again.\n");
        }
    }

    close(control_sock);
    return 0;
}

void dir_command(int control_sock,const char *path)
{
    char buffer[BUFFER_SIZE] = {0};
    char command[BUFFER_SIZE];
    memset(buffer,0,BUFFER_SIZE);
    snprintf(command, sizeof(command), "DIR %s", path);
    send(control_sock, command, strlen(command), 0);
    read(control_sock, buffer, BUFFER_SIZE);
    printf("%s",buffer);
    if(strcmp(buffer,Error_550)!=0)
    {
        printf("Directory listing:\n%s", buffer);
        memset(buffer,0,BUFFER_SIZE);
    }
}
```

```
void get_command(int control_sock, const char *filename)
{
   char command[BUFFER_SIZE];
   char buffer[BUFFER_SIZE] = {0};

   memset(buffer,0,BUFFER_SIZE);
   snprintf(command, sizeof(command), "GET %s", filename);
   send(control_sock, command, strlen(command), 0);
   read(control_sock, buffer, BUFFER_SIZE);
   if(strcmp(buffer,Error_550)==0)
   {
       printf("%s",buffer);
       return;
   }
   printf("%s",buffer);

   if (strncmp(buffer, "STARTING FILE TRANSFER\n", 24) == 0)
   {
       // Receive file over data socket
       int data_sock;
       struct sockaddr_in data_addr;

       data_sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
        data_addr.sin_family = AF_INET;
        data_addr.sin_port = htons(DATA_PORT);
        inet_pton(AF_INET, "127.0.0.1", &data_addr.sin_addr);
        connect(data_sock, (struct sockaddr *)&data_addr,
sizeof(data_addr));
        //read(control_sock, buffer, BUFFER_SIZE);
        //printf("%s",buffer);
        memset(buffer,0,BUFFER_SIZE);
        FILE *file = fopen(filename, "wb");  // Open file in binary mode
        if (file == NULL)
        {
            printf("Failed to create file\n");
            return;
        }
        int bytes_read;
        while ((bytes_read = read(data_sock, buffer, BUFFER_SIZE)) > 0)
        {
            fwrite(buffer, 1, bytes_read, file);
        }
        memset(buffer,0,BUFFER_SIZE);
        read(control_sock, buffer, BUFFER_SIZE);
        printf("%s\n", buffer);
        fclose(file);
        close(data_sock);
```

```
    }
}
```

Implementation Explanation-

1. Client-Side (`client.cpp`)

- **Control Socket Creation: The client creates a TCP control socket to communicate with the server.**
- **Connecting to Server: The client connects to the server's control channel on port 8080.**
- **User Command Input: The client reads commands from the user (`DIR`, `GET <filename>`, `EXIT`).**
- **DIR Command Handling: Sends `DIR` to the server, receives and prints a list of files.**
- **GET Command Handling:**
  - **Opens a separate data channel on port 8081.**
  - **Receives the requested file from the server in chunks and saves it locally.**
  - **Displays the time taken for file transfer using a timer.**
- **EXIT Command: Closes the control connection and exits the program.**

2. Server-Side (`server.cpp`)

- **Control Channel Setup: The server listens for client connections on the control channel (port 8080).**
- **Client Request Handling: For each client, the server processes commands (`DIR`, `GET <filename>`).**
- **DIR Command Handling:**
  - **Reads the `shared_files` directory and sends the file list to the client.**
- **GET Command Handling:**
  - **Creates a data socket to transfer the requested file over the data channel (port 8081).**
  - **Sends the file to the client in chunks, ensuring the entire file is transmitted.**
- **Error Handling: Handles cases where a file is not found or socket operations fail, sending appropriate error messages to the client.**

Overall Workflow:

1. **Client connects to the server on the control channel.**
2. **Client sends a command (`DIR` for listing or `GET <filename>` for file transfer).**
3. **Server processes the command:**
   - **For DIR, it sends the list of files.**
   - **For GET, it opens the data channel and transfers the file.**
4. **Client downloads the file and measures the transfer time.**

5. **Server continues to listen for more client requests.**

Exercise

PART-II

1. Implementing the Protocol with Multithreading

Multithreading Overview

- **Multithreading allows multiple clients to connect to the server simultaneously without blocking each other. A separate thread handles each client connection.**
- **This enhances the server's ability to manage concurrent file transfers, making it more efficient in handling requests.**

Code Implementation

1. **Server Code Changes:**
    - **Use `pthread_create` to spawn a new thread for each client connection.**
    - **Each thread handles the client's requests (DIR, GET) independently.**

c

Copy code

```
#include <pthread.h>
```

```
// ... other includes


void *handle_client(void *client_socket) {
    int sock = *(int *)client_socket;
    // Handle DIR and GET commands here
    // Close socket when done
    close(sock);
    return NULL;
}


int main() {
    // Set up socket, bind, listen, etc.

    while (1) {
        int client_sock = accept(server_sock, (struct
sockaddr *)&client_addr, &addr_len);
        pthread_t thread_id;
        pthread_create(&thread_id, NULL, handle_client,
(void *)&client_sock);
```

```
        pthread_detach(thread_id); // Detach thread to avoid
memory leaks
    }
    // Close server socket
}
```

2. **Client Code:**
    - **Remains largely unchanged; you can initiate multiple client instances to test the server's performance.**

Contrast with the Previous Implementation

- **Single-Threaded vs. Multithreaded:**
    - **Single-threaded: Each client request was handled sequentially. If one client was being served, others had to wait, leading to potential bottlenecks.**
    - **Multithreaded: Each client is handled in a separate thread, allowing multiple requests to be processed simultaneously, improving throughput and response times.**

Expected Performance Enhancements

- **Lower Completion Times: With multithreading, you should see reduced completion times for file transfers as the server can handle multiple clients concurrently.**

- **Scalability: The server can handle many more simultaneous connections without a significant increase in response time.**

Measuring Completion Time

- **As before, you would measure the time taken for file transfers, but now with varying numbers of clients.**

Expected Output

You can expect to collect data on completion times as you increase the number of clients:

Summary of Expected Output

- **Completion times will generally decrease with the use of multithreading, especially for small to moderate numbers of clients.**
- **For larger numbers of clients, the increase in completion time may be less severe compared to the single-threaded approach, showcasing the benefits of concurrent processing.**

Conclusion

By implementing multithreading, your file transfer protocol will become more robust and efficient, able to serve multiple clients simultaneously with better performance characteristics. This setup simulates real-world scenarios where servers often handle many requests at once, making it a valuable skill to develop in network programming.

In a multithreaded server handling file transfers, the impact on time can be analyzed in several key ways:

1. Reduced Wait Time

- **Single-threaded servers handle one client at a time. If one client's request takes time (e.g., due to a large file transfer), other clients must wait.**
- **Multithreaded servers allow each client to be served independently in a separate thread, significantly reducing the wait time for each client, especially under high load.**

2. Concurrent Processing

- **With multithreading, multiple file transfers can occur simultaneously. This means:**
  - **While one thread is busy reading a file from disk, another thread can be writing data to a different client.**
  - **This overlapping of operations helps utilize system resources (CPU, I/O) more effectively, leading to faster overall performance.**

3. Increased Throughput

- **The server can handle more requests per unit time. For example, if the server can process 10 requests per second in a single-threaded model, multithreading might allow it to handle 50 or more, depending on the system's capabilities.**

4. Impact of Resource Contention

- **As the number of threads increases, contention for shared resources (e.g., CPU, memory, network bandwidth) may arise:**
    - **Context Switching: If too many threads are created, the operating system may spend more time switching between threads than executing them, leading to increased completion times.**
    - **I/O Bottlenecks: If multiple threads are reading from or writing to the same disk, performance can degrade. However, this effect is typically outweighed by the benefits of concurrency until a saturation point is reached.**

5. Scalability

- **The ability to handle increasing numbers of clients effectively makes the server scalable. As you increase the number of clients from 1 to, say, 50:**
    - **Initial Increase in Time: You might see completion times rise slowly at first.**
    - **Threshold Effect: After a certain point, especially if resources are limited, you may see a sharper increase in time due to the reasons mentioned above.**

Example Measurement

Conclusion

Overall, multithreading generally leads to lower completion times for file transfers when managed well, allowing a server to efficiently handle multiple clients simultaneously. However, there are diminishing returns as resource contention increases, emphasizing the importance of balancing the number of threads with the available system resources.

```
vboxuser@bhoomishpatel:~/Desktop/lab_6$ g++ client_a.cpp -o client_a
vboxuser@bhoomishpatel:~/Desktop/lab_6$ ./client_a
Enter command (DIR,GET <filename>,EXIT):DIR ~Desktop/random
File transfer completed in: 0.0006845 seconds
Enter command (DIR,GET <filename>,EXIT):EXIT
vboxuser@bhoomishpatel:~/Desktop/lab_6$
```


```
vboxuser@bhoomishpatel:~/Desktop/lab_6$ g++ client_a.cpp -o client_a
vboxuser@bhoomishpatel:~/Desktop/lab_6$ ./client_a
Enter command (DIR,GET <filename>,EXIT):DIR ~Desktop/random
File transfer completed in: 0.0006845 seconds
Enter command (DIR,GET <filename>,EXIT):EXIT
vboxuser@bhoomishpatel:~/Desktop/lab_6$
```


Code for question 2

```
#include <iostream>

#include <thread>

#include <vector>

#include <string>
```

```
#include <fstream>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <dirent.h>


#define CONTROL_PORT 8080

#define DATA_PORT 8081


void handle_client_control(int control_socket, int data_socket) {

    char buffer[1024];

    while (true) {

        int bytes_received = recv(control_socket, buffer, sizeof(buffer),
0);

        if (bytes_received <= 0) {

            break;

        }

        buffer[bytes_received] = '\0';

        std::string command(buffer);


        if (command == "DIR") {

            DIR *dir;

            struct dirent *ent;
```

```
            std::string file_list;
            if ((dir = opendir(".")) != NULL) {
                while ((ent = readdir(dir)) != NULL) {
                    file_list += std::string(ent->d_name) + "\n";
                }
                closedir(dir);
            }
            send(control_socket, file_list.c_str(), file_list.size(), 0);
        } else if (command.find("GET") == 0) {
            std::string filename = command.substr(4);
            std::ifstream file(filename, std::ios::binary);
            if (file) {
                file.seekg(0, std::ios::end);
                size_t file_size = file.tellg();
                file.seekg(0, std::ios::beg);

                char file_buffer[1024];
                while (!file.eof()) {
                    file.read(file_buffer, sizeof(file_buffer));
                    send(data_socket, file_buffer, file.gcount(), 0);
                }
                file.close();
            } else {
                std::string error_msg = "Error: File not found";
```

```
                send(control_socket, error_msg.c_str(), error_msg.size(),
0);

            }

        }

    }

    close(control_socket);

    close(data_socket);

}


void client_handler(int client_control_socket) {

    int client_data_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in data_addr;

    data_addr.sin_family = AF_INET;

    data_addr.sin_port = htons(DATA_PORT);

    data_addr.sin_addr.s_addr = INADDR_ANY;


    connect(client_data_socket, (sockaddr*)&data_addr, sizeof(data_addr));


    handle_client_control(client_control_socket, client_data_socket);

}


int main() {

    int control_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in server_addr;
```

```
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(CONTROL_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    bind(control_socket, (sockaddr*)&server_addr, sizeof(server_addr));
    listen(control_socket, 5);

    std::vector<std::thread> threads;

    while (true) {
        int client_control_socket = accept(control_socket, NULL, NULL);
        threads.push_back(std::thread(client_handler,
client_control_socket));
    }

    for (auto& th : threads) {
        if (th.joinable()) {
            th.join();
        }
    }

    close(control_socket);
    return 0;
}
```

```cpp
#include <iostream>
#include <string>
#include  <fstream>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define CONTROL_PORT 8080
#define DATA_PORT 8081

void receive_file(int data_socket, const std::string& filename) {
    std::ofstream file(filename, std::ios::binary);
    char buffer[1024];
    int bytes_received;
```

```
    while ((bytes_received = recv(data_socket, buffer, sizeof(buffer), 0))
> 0) {
        file.write(buffer, bytes_received);
    }

    file.close();
}

int main() {
    int control_socket = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(CONTROL_PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(control_socket, (sockaddr*)&server_addr, sizeof(server_addr));

    int data_socket = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in data_addr;
    data_addr.sin_family = AF_INET;
    data_addr.sin_port = htons(DATA_PORT);
    data_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(data_socket, (sockaddr*)&data_addr, sizeof(data_addr));
```

```
    std::string command;
    while (true) {
        std::cout << "Enter command (DIR/GET filename): ";
        std::getline(std::cin, command);

        send(control_socket, command.c_str(), command.size(), 0);

        if (command == "DIR") {
            char buffer[1024];
            int bytes_received = recv(control_socket, buffer,
sizeof(buffer), 0);
            buffer[bytes_received] = '\0';
            std::cout << buffer;
        } else if (command.find("GET") == 0) {
            std::string filename = command.substr(4);
            receive_file(data_socket, filename);
        }
    }

    close(control_socket);
    close(data_socket);
    return 0;
}
```

1. Server Code

1. **Control and Data Socket Creation:**
    - **The server creates a control socket to listen for client commands on port `8080`.**
    - **For file transfers, a separate data socket is created, which connects to clients on port `8081`.**
2. **Client Handler:**
    - **The server runs a loop to accept client connections.**
    - **Each accepted connection is handled in a separate thread using `std::thread`, allowing multiple clients to be served concurrently.**
    - **For each client, the server establishes a control connection to process commands and a data connection for file transfers.**
3. **Handling Client Commands:**
    - **The server reads client commands from the control socket.**
    - **If the command is `DIR`, it lists all the files in the current directory and sends the list back to the client.**
    - **If the command is `GET <filename>`, the server reads the requested file in chunks from the file system and sends it over the data socket to the client.**
    - **If a file isn't found, an error message is sent back.**
4. **Multithreading:**
    - **Each client is handled in its own thread. The `client_handler` function manages the communication with the client through both the control and data channels, ensuring that file transfers and directory listings happen concurrently for multiple clients.**

---

2. Client Code

1. **Control and Data Socket Creation:**

- The client creates a control socket to send commands (like `DIR` or `GET`) to the server on port `8080`.
- A separate data socket is created on port `8081` to receive files from the server when requested.

2. Sending Commands:
   - The user inputs commands (`DIR` or `GET <filename>`).
   - The client sends these commands to the server via the control socket.
3. Receiving Directory Listing:
   - If the user sends the `DIR` command, the server responds with a list of files in the directory.
   - The client receives and displays this list.
4. Receiving Files:
   - For the `GET <filename>` command, the client requests a specific file from the server.
   - The file is transferred over the data socket in chunks and written to a local file.
   - The client receives the file from the server and saves it using the `receive_file` function, which writes the data to the disk.

Key Points:

- Concurrency: The server can handle multiple clients simultaneously through the use of threads.
- Separation of Control and Data Channels: Control commands (`DIR`, `GET`) are handled on one socket, while file transfers are performed on a separate socket to avoid blocking.
- Modular Design: File handling, directory listing, and command processing are handled in dedicated functions, making the code more readable and maintainable.