File Transfer over TCP/IP in Command Line Interface (CLI)

Hoang Minh Quan - 23BI14371

November 26, 2025

1 Goal

The goal of this practical work is to implement a 1–1 file transfer over TCP/IP using a Command Line Interface (CLI). The system is composed of:

- one TCP server,
- one TCP client,
- communication over IPv4 sockets on localhost.

The implementation language is C, compiled and executed on Kali Linux with gcc.

2 Protocol Design

The protocol is intentionally simple. The client and server use a fixed TCP port (8080) and a fixed IP address (127.0.0.1). No application-level headers are sent:

- the client opens file example_file.txt and sends its content as a byte stream
- the server receives bytes on the socket and writes them to received_file.txt
- the end of file transfer is detected when the client closes the connection and the server's recv() returns 0.

Figure 1 describes the client–server interaction from socket creation to file transfer completion.

At a higher level, the communication can be viewed as the client connecting to the server through the network, sending the file, and closing the connection (Figure 2).

3 System Organization

The system is organized into two programs: server and client. Both run in the same directory /home/kali/Documents/Practical Work/Practical Work 1.

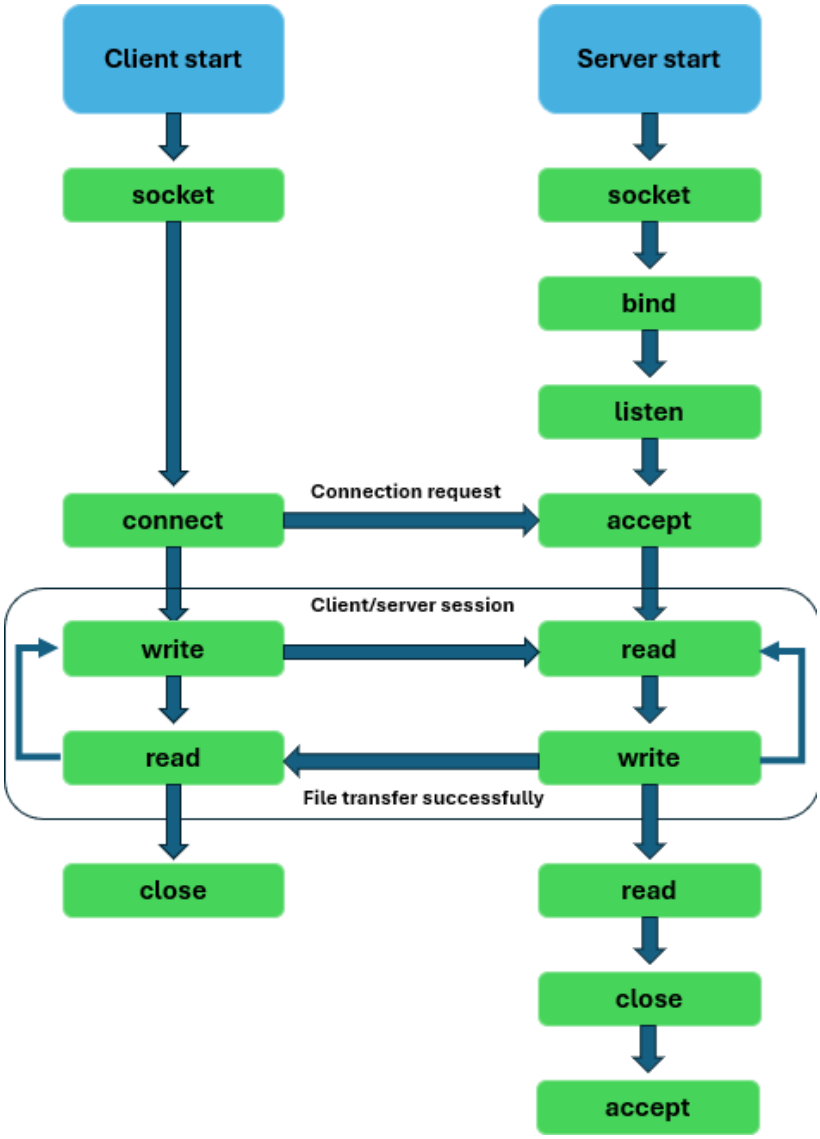

Figure 1: Client–server TCP file transfer flow (socket, connect, write/read, close).

Server

The server performs the following steps:

1. create a TCP socket (socket());
2. bind the socket to IP 127.0.0.1 and port 8080 (bind());
3. listen for incoming connections (listen());
4. accept a single connection (accept());
5. receive file data using recv() and write it to received_file.txt;
6. close the connection.

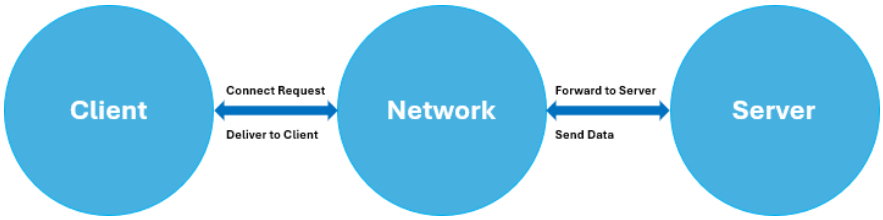

Figure 2: High-level architecture: Client–Network–Server.

Client

The client performs these steps:

1. create a TCP socket;
2. connect to 127.0.0.1:8080 (connect());
3. open example_file.txt in read mode;
4. send file content to the server using send();
5. close the socket when finished.

4 Implementation Details

4.1 Server Code (C)

Listing 1: Server program (server.c)

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

#define SIZE 1024

void write_file(int sockfd)
{
    int n;
    FILE *fp;
    char *filename = "received_file.txt";
    char buffer[SIZE];

    fp = fopen(filename, "w");
    if(fp==NULL)
    {
        perror("Error in creating file.");
        exit(1);
    }
```

```
    while(1)
    {
        n = recv(sockfd, buffer, SIZE, 0);
        if(n<=0)
        {
            break;
            return;
        }
        fprintf(fp, "%s", buffer);
        bzero(buffer, SIZE);
    }
    return;

}

int main ()
{
    char *ip = "127.0.0.1";
    int port = 8080;
    int e;

    int sockfd, new_sock;
    struct sockaddr_in server_addr, new_addr;
    socklen_t addr_size;
    char buffer[SIZE];
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd<0)
    {
        perror("Error in socket");
        exit(1);
    }
    printf("Server socket created. \n");

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = port;
    server_addr.sin_addr.s_addr = inet_addr(ip);

    e = bind(sockfd,(struct sockaddr*)&server_addr, sizeof(server_addr));
    if(e<0)
    {
        perror("Error in Binding");
        exit(1);
    }
    printf("Binding Successfull.\n");

    e = listen(sockfd, 10);
    if(e==0)
    {
        printf("Listening...\n");
    }
    else
    {
        perror("Error in Binding");
        exit(1);
    }
    addr_size = sizeof(new_addr);
    new_sock = accept(sockfd,(struct sockaddr*)&new_addr, &addr_size);

    write_file(new_sock);
    printf("Data written in the text file ");
}
```

4.2 Client Code (C)

Listing 2: Client program (client.c)

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

#define SIZE 1024

void send_file(FILE *fp, int sockfd)
{
    char data[SIZE] = {0};

    while(fgets(data, SIZE, fp)!=NULL)
    {
        if(send(sockfd, data, sizeof(data), 0)== -1)
        {
            perror("Error in sendung data");
            exit(1);
        }
        bzero(data, SIZE);
    }
}

int main()
{
    char *ip = "127.0.0.1"; //Change to server address to send file in need
    int port = 8080;
    int e;

    int sockfd;
    struct sockaddr_in server_addr;
    FILE *fp;
    char *filename = "example_file.txt";
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd<0)
    {
        perror("Error in socket");
```

```
        exit(1);
    }
    printf("Server socket created. \n");

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = port;
    server_addr.sin_addr.s_addr = inet_addr(ip);

    e = connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
    if(e == -1)
    {
        perror("Error in Connecting");
        exit(1);
    }
    printf("Connected to server.\n");
    fp = fopen(filename, "r");
    if(fp == NULL)
    {
        perror("Error in reading file.");
        exit(1);
    }
    send_file(fp,sockfd);
    printf("File data send successfully. \n");
    close(sockfd);
    printf("Disconnected from the server. \n");
    return 0;

}
```

5 Execution and Testing

All experiments are run on Kali Linux. The source files are compiled with gcc and the input file is created using the shell:

```
gcc server.c -o server
gcc client.c -o client
echo "Hello from client" > example_file.txt
```

Figure 3 shows the compilation commands and the creation of example_file.txt.

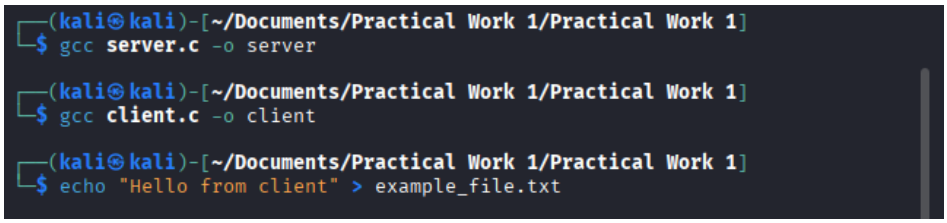

Figure 3: Compiling the C programs and creating the input file.

The server is started in one terminal and the client in another (Figure 4). The log messages confirm that the connection is established and that the file is sent and written successfully.

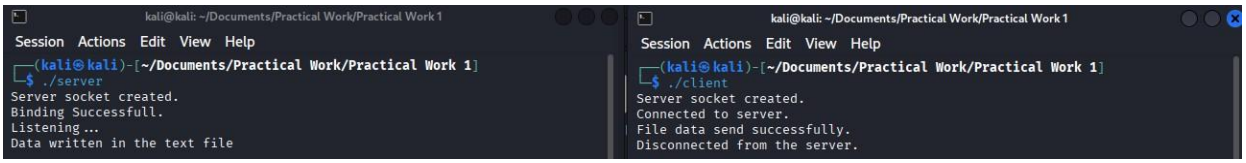

Figure 4: Server and client running in two terminals on Kali Linux.

The working directory after execution contains the compiled binaries and both text files (example_file.txt and received_file.txt), as shown in Figure 5.

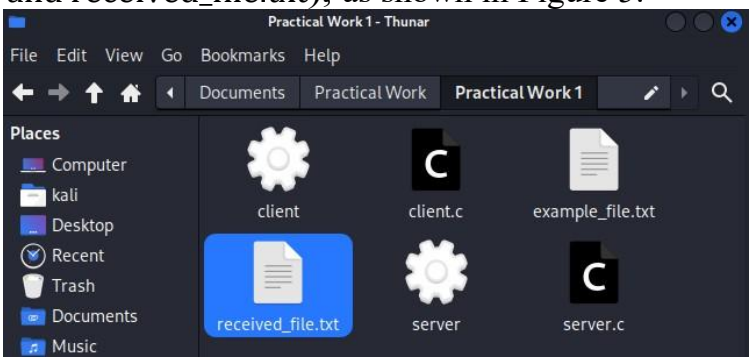

Figure 5: Files in the project directory after a successful transfer.

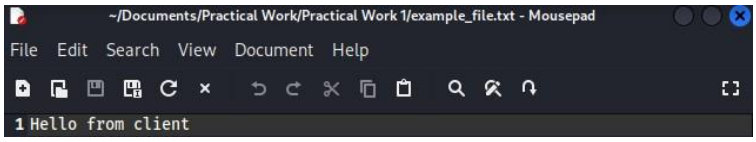

Figure 6: Content of example_file.txt(client side).

Figure 6 shows the content of example_file.txt on the client side.
Finally, Figure 7 shows the network configuration (ip r) confirming that both processes run on the same host.

Figure 7: IP routing table on Kali Linux used during testing.

6 Conclusion

In this practical work, a minimal but functional TCP file transfer system was implemented in C on Kali Linux. The client and server programs use the standard BSD socket API to establish a connection over 127.0.0.1:8080 and transfer a text file from client to server.

The tests show that the design and implementation correctly achieve the goal of one-to-one file transfer over TCP/IP. Possible extensions include adding support for variable filenames, binary files, error codes, and multi-client handling.