# Assignment-1

**Computer Networks**

BSDS-1

|  |  |
| --- | --- |
| Members | CMS |
| M Muntazar | 470861 |
| M Wasif Shakeel | 456092 |

**Link of the Video Recording:**

<https://private-user-images.githubusercontent.com/66616578/376018132-1c428e71-fd7f-4308-bcc1-09f9a9bf4133.mp4?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..K-27le2x0d7DCQwDqXUXv85A_5DLWQwWZ5Nl66A27FA>

**Alternate Link:**  
<https://github.com/Overproness/Computer-Networks-Assignment-1/blob/main/README.md>

<https://github.com/Overproness/Computer-Networks-Assignment-1>

## **Server Side Code:**

#include <arpa/inet.h>

#include <netinet/in.h>

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#define PORT\_NUMBER 8877 // Port number on which the server listens for connections

#define BUFFER\_SIZE 1024 // Maximum size of the buffer for reading data

int main()

{

    // Step 1: Define variables for socket setup and initialize them

    struct sockaddr\_in server\_addr, client\_addr;     // Structures to hold the server and client address info

    socklen\_t client\_addr\_len = sizeof(client\_addr); // Length of the client address structure

    int server\_socket, client\_socket;                // Sockets for listening and accepting client connections

    // Zero out the server address structure and configure it

    memset(&server\_addr, 0, sizeof(server\_addr));    // Clear the server address memory

    server\_addr.sin\_family = AF\_INET;                // Use IPv4 addressing

    server\_addr.sin\_port = htons(PORT\_NUMBER);       // Set the port number (convert to network byte order)

    server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY); // Allow connections from any IP address

    // Step 2: Create a listening socket

    if ((server\_socket = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)

    {

        perror("Error: Unable to create listening socket."); // Print error if socket creation fails

        return 1;

    }

    // Step 3: Set socket options to allow address reuse (prevents binding errors on restart)

    int socket\_option = 1;

    if (setsockopt(server\_socket, SOL\_SOCKET, SO\_REUSEADDR, &socket\_option, sizeof(socket\_option)) < 0)

    {

        perror("Error: Unable to set socket options."); // Print error if setting socket options fails

        return 1;

    }

    // Step 4: Bind the socket to the specified address and port

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

    {

        perror("Error: Socket binding failed."); // Print error if binding fails

        return 1;

    }

    // Step 5: Start listening for incoming client connections

    if (listen(server\_socket, 16) < 0)

    {

        perror("Error: Unable to listen for incoming connections."); // Print error if listening fails

        return 1;

    }

    // Step 6: Accept an incoming connection from a client

    printf("Server is waiting for a client to connect...\n");

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

    {

        perror("Error: Failed to accept client connection."); // Print error if accepting connection fails

        return 1;

    }

    // Notify about successful client connection and display client's IP address

    printf("Client connected successfully with IP address: %s\n", inet\_ntoa(client\_addr.sin\_addr));

    // Step 7: Open the video file to be sent to the client

    int video\_file\_fd = open("testVideo.mp4", O\_RDONLY); // Open the video file in read-only mode

    if (video\_file\_fd < 0)

    {

        perror("Error: Unable to open video file for reading."); // Print error if file cannot be opened

        close(client\_socket);                                    // Close the client socket on error

        return 1;

    }

    // Step 8: Prepare to read data from the file and send it to the client

    char data\_buffer[BUFFER\_SIZE]; // Buffer to hold file data

    ssize\_t read\_bytes;            // Variable to store the number of bytes read from the file

    // Step 9: Read the file and send the data in chunks to the client

    while ((read\_bytes = read(video\_file\_fd, data\_buffer, sizeof(data\_buffer))) > 0)

    {

        send(client\_socket, data\_buffer, read\_bytes, 0); // Send data chunk to the client

    }

    // Step 10: Send end-of-file (EOF) indicator to signal file transmission is complete

    const char \*eof\_message = "EOF";                          // End of file message

    send(client\_socket, eof\_message, strlen(eof\_message), 0); // Send EOF message to the client

    // Step 11: Close the file and sockets after file transmission is done

    close(video\_file\_fd); // Close the file descriptor for the video file

    close(client\_socket); // Close the client socket

    close(server\_socket); // Close the listening socket

    // Notify that the file was sent and connection has been closed

    printf("File transfer completed successfully. Connection closed.\n");

    return 0; // Return success code

}

## **Client Side Code:**

#include <arpa/inet.h>

#include <netinet/in.h>

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#define PORT\_NUMBER 8877 // Server port number for connection

#define BUFFER\_SIZE 1024 // Maximum buffer size for data transfer

int main()

{

    // Step 1: Define and initialize variables for socket setup

    struct sockaddr\_in server\_addr; // Structure to hold the server's address information

    int client\_socket;              // Variable to store the client's socket descriptor

    // Zero out the server address structure and configure it

    memset(&server\_addr, 0, sizeof(server\_addr));    // Clear the memory for the server address

    server\_addr.sin\_family = AF\_INET;                // Use IPv4 addressing

    server\_addr.sin\_port = htons(PORT\_NUMBER);       // Set the port number (convert to network byte order)

    server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY); // Allow connections from any IP address (localhost)

    // Step 2: Create a socket for communication

    if ((client\_socket = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)

    {

        perror("Error: Unable to create socket."); // Print error if socket creation fails

        return 1;

    }

    // Step 3: Establish connection to the server using the configured server address

    if (connect(client\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

    {

        perror("Error: Connection to the server failed."); // Print error if connection attempt fails

        return 1;

    }

    // Step 4: Open a new file to save the received video data

    int video\_file\_fd = open("downloadedVideo.mp4", O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH);

    if (video\_file\_fd < 0)

    {

        perror("Error: Unable to create or open the video file for writing."); // Print error if file can't be opened

        close(client\_socket);                                                  // Close the socket before exiting due to error

        return 1;

    }

    // Step 5: Prepare to receive data from the server

    char data\_buffer[BUFFER\_SIZE]; // Buffer to store incoming data

    ssize\_t received\_bytes;        // Variable to store the number of bytes received in each segment

    // Step 6: Loop to receive the video file in chunks

    while (true)

    {

        received\_bytes = recv(client\_socket, data\_buffer, sizeof(data\_buffer), 0); // Receive data from server

        if (received\_bytes < 0)

        {

            perror("Error: Problem occurred while receiving data."); // Print error on data reception failure

            break;

        }

        else if (received\_bytes == 0)

        {

            // Server has closed the connection

            break;

        }

        // Step 7: Check if the server sent the end-of-file (EOF) indicator

        if (strncmp(data\_buffer, "EOF", received\_bytes) == 0)

        {

            printf("Notification: End of video file transmission detected.\n"); // Notify user that EOF was reached

            break;

        }

        // Step 8: Write the received chunk of data into the video file

        write(video\_file\_fd, data\_buffer, received\_bytes); // Save the received data to the file

    }

    // Step 9: Cleanup - Close the file and socket after data reception is complete

    close(video\_file\_fd); // Close the video file descriptor

    close(client\_socket); // Close the socket connection to the server

    // Final notification to the user

    printf("Success: Video file has been successfully received, and the connection is now closed.\n");

    return 0; // Return success code

}

## **Client-Side Video Receiver**

### Approach:

The first piece of code is a client-side program responsible for connecting to a server, receiving a video file from it, and saving the file locally. The process involves socket programming to establish communication between the client and the server, file I/O operations to write the received data into a file, and a simple protocol to detect when the file transfer is complete (using an "EOF" message).

### Design:

1. **Socket Setup:**
   * The program uses socket() to create a TCP (stream) socket that communicates with a server using IPv4 (AF\_INET). The connect() function attempts to connect to the server, which is running on a specific port defined by PORT\_NUMBER.
2. **File Creation:**
   * After successfully connecting to the server, the program attempts to open a local file called "downloadedVideo.mp4" in write mode. If the file cannot be opened, the program gracefully terminates by closing the socket.
3. **Receiving Data:**
   * The program uses a buffer of fixed size (BUFFER\_SIZE) to receive chunks of data from the server using recv(). The chunks are continuously written to the opened video file.
4. **End-of-File Handling:**
   * The program checks if the received data contains the "EOF" marker to signify the end of the video file transfer. If "EOF" is detected, the program breaks out of the loop.
5. **Cleanup:**
   * Once the transfer is complete, the client closes both the file and the socket, ensuring a clean exit.

### Description:

This code represents a TCP client that receives video file data from a remote server. The client connects to the server using a predefined port and saves the transmitted file locally. The program is designed to handle errors such as connection failure, file opening issues, and data reception problems. An "EOF" marker indicates the end of the file transfer, after which the connection is closed.

## **Server-Side Video Sender**

### Approach:

The second piece of code is a server-side program responsible for accepting incoming client connections, reading a video file from the local system, and sending it to the client in chunks. The server uses TCP sockets to listen for connections, and after a client connects, it sends the contents of the video file to the client until the entire file is transmitted.

### Design:

1. **Socket Setup:**
   * The server sets up a TCP socket using socket() with the IPv4 address family (AF\_INET) and the stream protocol (SOCK\_STREAM). The bind() function assigns the server's IP address and a port number (PORT\_NUMBER), while listen() enables the server to accept incoming connections.
2. **Accepting Client Connection:**
   * Once the server is listening for connections, it calls accept() to establish a connection with a client. This blocks the server until a client initiates a connection. The client's IP address is printed once the connection is accepted.
3. **File Handling:**
   * The server opens a video file ("sampleVideo.mp4") in read-only mode. If the file cannot be opened, the server closes the client connection and exits.
4. **Data Transmission:**
   * Using a buffer of fixed size (BUFFER\_SIZE), the server reads chunks of data from the file and sends them to the client using send(). This loop continues until the entire video file is sent.
5. **End-of-File Handling:**
   * After sending all the video data, the server sends an "EOF" message to the client, indicating the completion of the file transfer.
6. **Cleanup:**
   * After the file transmission is done, the server closes the file descriptor and both the client and listening sockets.

### **Description:**

This code represents a simple TCP server that waits for a client to connect and then transmits a video file to the client. The server listens on a predefined port, accepts a connection from a client, reads the video file in chunks, and sends the data to the client. After transmitting the entire file, it signals the client using an "EOF" marker and gracefully closes the connection. The server is designed to handle basic errors, such as file opening failures or socket binding issues.

## **Summary of Key Differences:**

* **Client vs. Server:**
  + The first code acts as the **client**, responsible for receiving and saving the video file.
  + The second code is the **server**, responsible for sending the video file.
* **File I/O:**
  + The client writes the received file data to disk, while the server reads a file from disk to send to the client.
* **Network Communication:**
  + Both codes use TCP sockets to facilitate communication, but the client initiates the connection, while the server listens and accepts incoming connections.
* **EOF Handling:**
  + Both sides use an "EOF" message to mark the end of the file transmission, ensuring the client knows when the file transfer is complete.