Simulation of TCP using C programming

Manav Chokshi
19BEC068
Dept. of Electronics and Communication
Institute of Technology, Nirma University
Ahmedabad, India
19bec068@nirmauni.ac.in

Riya Gautam
20BEC038
Dept. of Electronics and Communication
Institute of Technology, Nirma University
Ahmedabad, India
20bec038@nirmauni.ac.in

***Abstract*—This report delves into the implementation and simulation of Transmission Control Protocol (TCP) using C programming. It explores the fundamental principles and inner workings of TCP, simulating its behavior in various network scenarios. Through C programming, the study investigates packet transmission, 3-way handshaking, and Data transfer for File Server Applications. The report justifies the use of mechanisms provided by the Operating System which are extensively used in implementation for synchronization and serialization between multiple processes in action.**

I. Introduction

The Transmission Control Protocol (TCP) stands as a cornerstone in TCP/IP protocol suite, ensuring reliable and ordered data transfer between interconnected devices. This report embarks on an exploratory journey into the intricate world of TCP, focusing on its implementation and simulation through the prism of C programming. The endeavor is to unravel the core principles and inner workings of TCP, exposing its behavior in a spectrum of diverse network scenarios.

With a focus on packet transmission, 3-way handshaking, and data transfer for File Server Applications, this report delves into the intricacies of these fundamental elements. Additionally, the report investigates the rationale behind utilizing mechanisms provided by the Operating System, pivotal in the implementation process. These mechanisms serve in synchronization and serialization, crucial for facilitating seamless communication among multiple active processes.

This exploration aims not only to reveal the fundamental features of TCP but also to underscore the practical implications and applications of these mechanisms in real-world network environments.

II. Transmission Control Protocol

A. Overview

TCP, a core protocol of the Internet Protocol Suite, operates at the transport layer and facilitates reliable, ordered, and error-checked delivery of data between network-connected devices. It underpins many applications such as web browsing, email, file transfer, and more. TCP breaks data into smaller packets for transmission, ensuring they are reliably delivered and reassembled in the correct order at the destination.

B. Connection Establishment and Termination

The TCP protocol uses a three-way handshake for connection establishment, ensuring both ends are ready to communicate. During this process, the client and server exchange SYN (synchronize) and ACK (acknowledgment) packets, confirming their readiness to send and receive data.

Connection termination involves a four-way handshake to ensure all data has been reliably exchanged and both ends are prepared to close the connection.

C. Reliability and Flow Control

TCP achieves reliability through various mechanisms, including sequence numbers, acknowledgments, and retransmissions. Each packet is assigned a sequence number, allowing the receiver to reassemble them in the correct order. Upon receiving data, the receiver sends acknowledgments, and the sender retransmits any unacknowledged packets to ensure reliable data transmission.

Flow control manages the data flow between sender and receiver, preventing overwhelming the receiver with a flood of data. This control is managed through window size adjustments, where the receiver informs the sender of its buffer space to regulate the data transmission rate.

D. Congestion Control

TCP handles congestion control to prevent network overload. It employs various algorithms like Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. These mechanisms monitor network conditions, adapt transmission speeds, and respond to packet loss to prevent network congestion, ensuring efficient data delivery.

III. 3-way handshaking

In Transmission Control Protocol (TCP), the 3-way handshaking is a fundamental process for establishing a connection between a client and a server. It ensures a reliable and orderly initiation of communication, essential for data exchange.

The process unfolds in three steps:

1) **SYN (Synchronize) from Client:** The client initiates the connection by sending a TCP segment with the SYN flag set in the control flags field and the source and destination ports specified in the TCP header. This segment contains a sequence number to begin the communication and a Header Length (HLEN) field indicating the size of the TCP header.
2) **SYN-ACK (Synchronize-Acknowledgment) from Server:** Upon receiving the SYN segment, the server responds with a TCP segment with both the SYN and ACK flags set in the control flags field of the TCP header. This segment acknowledges the client's request and includes the server's sequence number, acknowledgment number, and the Header Length.
3) **ACK (Acknowledgment) from Client:** Finally, the client acknowledges the server's response by sending a TCP segment with the ACK flag set in the control flags field of the TCP header. This packet also includes the Header Length and the acknowledgment number equal to the server's sequence number plus one.

The TCP header segments for each step might appear as follows:

```
Step 1: Client --> Server
Source Port: 1234
Destination Port: 80
HLEN: 20 bytes
Control Flags: SYN
Sequence Number: 100

Step 2: Server --> Client
Source Port: 80
Destination Port: 1234
HLEN: 20 bytes
Control Flags: SYN, ACK
Sequence Number: 300
Acknowledgment Number: 101

Step 3: Client --> Server
Source Port: 1234
Destination Port: 80
HLEN: 20 bytes
Control Flags: ACK
Acknowledgment Number: 301
```

This 3-way exchange, depicted in the TCP header segments, ensures that both ends (client and server) are synchronized and ready to communicate. It ensures a reliable and orderly connection setup before data transmission begins. Once this handshake is completed, data transfer can occur between the two entities over a secure channel.

IV. Methodology

To emulate the links between various processes, FIFOs are used which provide powerful mechanism of serialization and inherently synchronize the read and write operation. Hence, explicit synchronization is not required making the implementation less complex. All kinds of requests are passed in form of C structures via FIFO. Since FIFOs are unidirectional, 2 FIFOs are required for duplex connection between processes. All traffic from *Application Processes to TCP Process* is transmitted via FIFO called 'system'. This means the TCP process must **read** from the *system* FIFO whereas Application processes **write** to *system* FIFO.

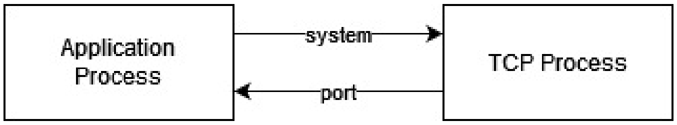

Fig. 1. 2 way link between Application Process and TCP process using FIFOs

For routing the Data/Control information from the TCP process to the respective Client Application, *port* FIFOs are used. These FIFOs are named based on the port application processes are listening to.

Two TCP processes on two different hosts are communicating via *Network*, and FIFOs are used to establish a 2-way link between the host machines.

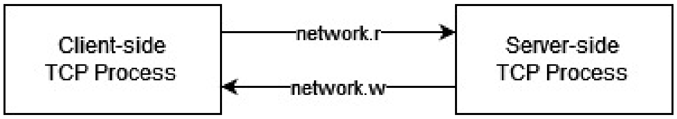

Fig. 2. 2-way link between Client TCP and Server TCP using FIFOs

This sums up the Interprocess Communication Part. Now the information that is being exchanged between the Application process and TCP process in form *packets*, are in the form of structures. There are 3 data structures used to transfer information.

1) TCP to TCP: `tcp_header`
2) TCP to Application Process: `tcpClientPacket`
3) Application to TCP: `clientReqPacket`

A. *tcp_header Structure*

This structure represents the TCP Header.

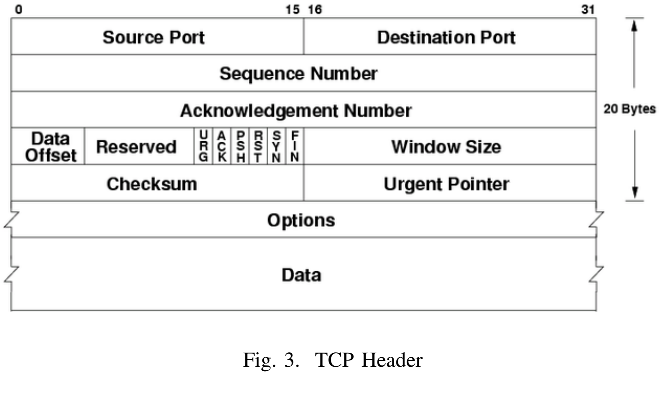

Fig. 3. TCP Header

B. *tcpClientPacket*

Structure which is transferred via port FIFOs.

TABLE I
TCP TO CLIENT PACKET

Component	Description
Data	Field containing Data
isData	Flag which is asserted when containing Data
info	Field containing Connection Status

C. *clientReqPacket*

Structure which is transferred via system FIFO.

TABLE II
CLIENT TO TCP PACKET

Components	Description
Source Port	For TCP Client to handle Connections
Dest Port	”. . . ”
isData	Flag which is asserted when containing Data
request	Field containing Connection Request
Data	Field containing Data Payload

V. IMPLEMENTATION

The complete program involves 4 processes:

1) Client-side TCP process
2) Server-side TCP process
3) Client Application requesting services from Server.
4) Server Application delivering services to Client.

Client-side TCP Process It goes live and waits for the Active Client Process requesting TCP service. If active client request is received then the process handles the client request.

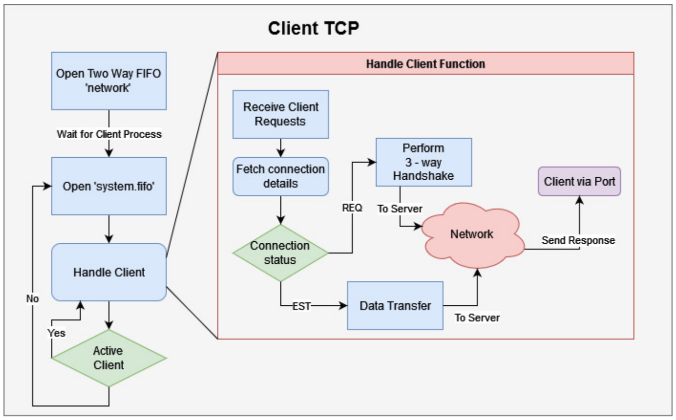

Fig. 4. Client-side TCP process

Server-side TCP Process It goes live and waits for the Active Server Application to listen to incoming connections. If an active server application are sensed, the TCP listens to the incoming packets from *network* fifo.

Server Application As server application, refer Fig. 6, File server is deployed which takes *file path* as an input and returns contents of the file if file exist.

Client Application Here, as a Client Application, refer Fig. 7, a process requesting for file is deployed where it takes the file path from the user and requests the file server to provide the file.

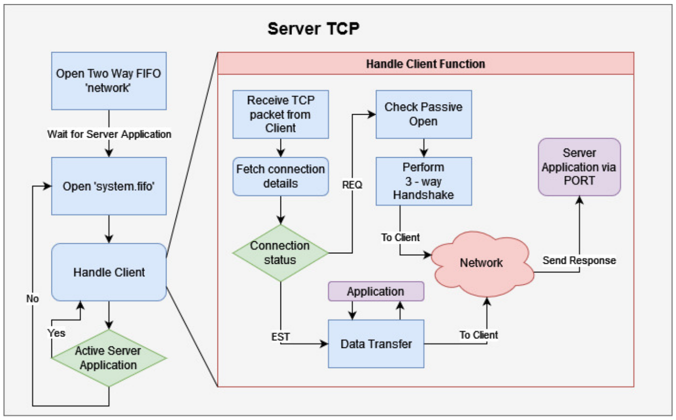

Fig. 5. Server-side TCP process

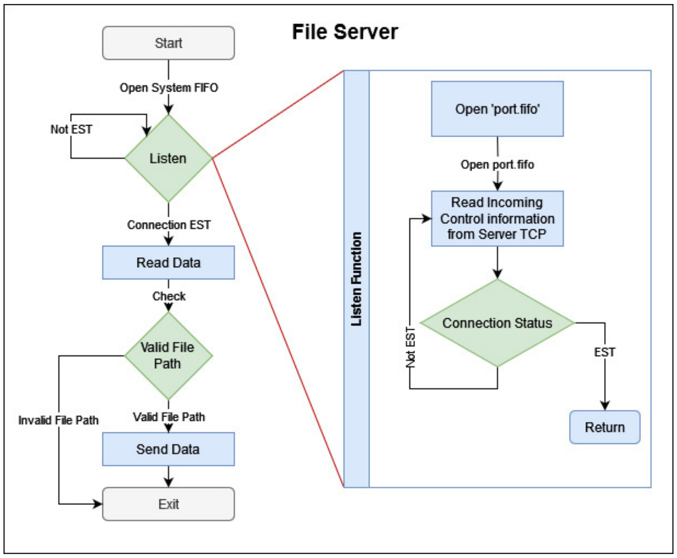

Fig. 6. Server Application process

VI. CONCLUSION

The exploration and implementation of Transmission Control Protocol (TCP) using C programming depicted a comprehensive understanding of the intricacies of this fundamental networking protocol. Through this venture, the core principles and operational mechanisms of TCP were dissected, shedding light on its functionalities and vital role in facilitating reliable and ordered data transfer between networked devices.

The report successfully highlighted key elements of TCP, encompassing packet transmission, the 3-way handshaking process, and data transfer for File Server Applications. Moreover, it justified the utilization of Operating System mechanisms, crucial for synchronization and serialization between multiple processes. These mechanisms were revealed as instrumental in ensuring seamless communication among active processes without the need for explicit synchronization.

The detailed discussion on the 3-way handshaking process elucidated its significance in establishing a robust and ordered connection between client and server entities. The exchange

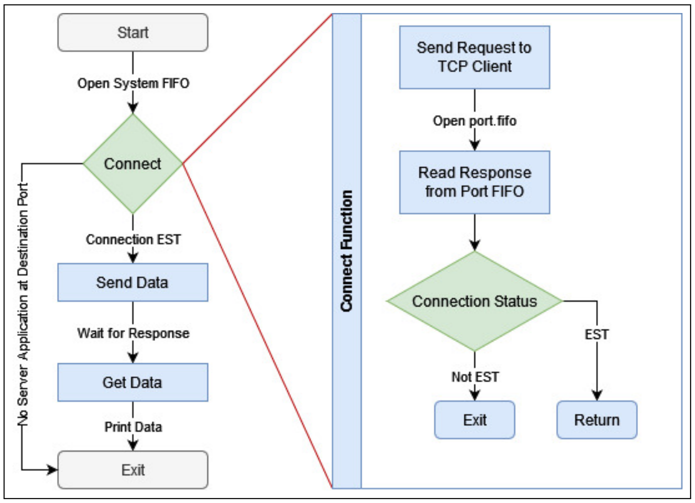

Fig. 7. Client Application process

of SYN, SYN-ACK, and ACK packets was articulated as a fundamental procedure ensuring the readiness of both ends for secure communication.

Furthermore, the methodology section delved into the intricacies of implementing TCP using C programming, employing FIFOs for interprocess communication. The structural representation of TCP headers and packets elucidated the exchange of information and control between various processes.

The comprehensive analysis of the implementation involved four distinct processes: the client-side TCP process, server-side TCP process, server application, and client application. Each process's role in initiating, managing, and facilitating communication was thoroughly described, providing a comprehensive overview of the complete system.

In conclusion, this report not only unveiled the core functionalities and intricacies of TCP but also provided a practical understanding of its implementation using C programming. It emphasized the significance of TCP in network communication and showcased the practical implications of employing these protocols in real-world scenarios. The methodologies and processes outlined in this exploration can serve as a foundation for understanding and developing network applications and protocols using C programming, contributing to the broader understanding of networking principles and their applications.

References

[1] Tcp/Ip Protocol Suite by Behrouz A. Forouzan
[2] UNIX Network Programming, Volume 2, Second Edition by W. Richard Stevens

Appendix A

Source Code

A.1 Client-side TCP Process

```
#include "tcp.h"

#define CLIENT_PORT_PATH "clientSide/ports/"
#define SYSTEM_FIFO_PATH "clientSide/system.fifo"

connectionList* head = NULL;
sem_t lock;

// File Descriptors of Network and System
int network_fd[2];
int system_fd;

connectionList* getConnections(uint16_t sourcePort, uint16_t destinationPort){
    if(head == NULL){
        head = (connectionList*)malloc(sizeof(connectionList));
        head->source_port = sourcePort;
        head->dest_port = destinationPort;
        head->connectionStatus = REQ;
        return head;
    }
    else{
        connectionList* curr = head;
        connectionList* prev = NULL;
        while(curr){
            if(curr->source_port == sourcePort && curr->dest_port == destinationPort){
                return curr;
            }
            prev = curr;
            curr = curr->next;
        }
        // printf("New Connection\n");
        // Adding new node at last and returning it for the handler to update the details
        connectionList* newNode = (connectionList*)malloc(sizeof(connectionList));
        newNode->source_port = sourcePort;
        newNode->dest_port = destinationPort;
        newNode->connectionStatus = REQ;
        prev->next = newNode;
        return newNode;
    }
}

int hanshake(uint16_t sourcePort, uint16_t destinationPort){
    /*
        -> handshake() performs 3-way handshaking for requested connection on (sourcePort,destPort).
        -> Returns either EST(0) connection or RST(1) connection in case no passive open on Destination Port.
    */

    tcp_header syn_packet, ack_syn_packet, ack_packet;

    // Received a connection request from client process
    // Assign a random sequence number

    int client_seqnum = rand();

    // Make SYN segment
    syn_packet = create_tcp_header(sourcePort, destinationPort, client_seqnum, 0, SYN_PACKET, 0, 0, 0);
    // Send Packet
    send_packet(network_fd[1], &syn_packet, sizeof(syn_packet));
    printf("SYN_PACKET Sent!\n");
    printTcpHeader(&syn_packet);



    // Receive ACK+SYN segment
    receive_packet(network_fd[0], &ack_syn_packet, sizeof(ack_syn_packet));
    if(ack_syn_packet.data_offset_flags == ACK_SYN_PACKET){
        printf("Received ACK+SYN PACKET\n");
        printTcpHeader(&ack_syn_packet);

        if(ack_syn_packet.ack_num == syn_packet.sequence_num + 1){
            // Opent Connection for Client
            // Print ACK received for the Client.
            // Send ACK packet for server side SYN request
            copyTcpHeader(&ack_packet, &ack_syn_packet);
            ack_packet.sequence_num = syn_packet.sequence_num;
            ack_packet.ack_num = ack_syn_packet.sequence_num + 1;
            ack_packet.data_offset_flags = ACK_PACKET;
        }
    }
    else{
        // Other than ACK
        printf("Received RST segment");
        if(ack_syn_packet.data_offset_flags == RST_PACKET){
            printf("No Server Application at Destination Port: %d", destinationPort);
            return 1;
        }
    }


    // Send ACK packet
    printf("Sent ACK PACKET\n");
    printTcpHeader(&ack_packet);
    send_packet(network_fd[1], &ack_packet, sizeof(ack_packet));

    return 0;
}

void handleClient(void){
    printf("Handling Clients.\n");
    while(true){
        clientReqPacket* reqPacket = (clientReqPacket*)malloc(sizeof(clientReqPacket));

        // Read returns data on the FIFO, if no data but write ends are open then shall block the process until any data is written on FIFO.
        ssize_t readBytes = Read(system_fd, reqPacket, sizeof(clientReqPacket));
        if(readBytes == 0){
            // If all write ends are closed (No client Process), 'read' returns 0 -> EOF.
            printf("No Active Clients.\n");
            close(system_fd);
            // Close the FIFO read end.
            // Go back to main function and Open Read end again. Open will again block the process execution until a client process open FIFO for Write.
            return;
        }
        printf("Received:\n");
        print_clientReqPacket(reqPacket);

        // Get Records of Current Request, if not then create new.
        connectionList* activeReq = getConnections(reqPacket->source_port, reqPacket->dest_port);
        // Printing Connection Record
        print_connectionList(activeReq);

        // Perform action based on request received.
        if(reqPacket->isData == false){
            printf("Connection Control..\n");
            // Client Process can either request to OPEN connection or CLOSE connection
            if(reqPacket->request == REQ){
                // Open FIFO end for Writing, store the file descriptor in 'connectionList->portFd'
                // Perform Handshake
                // Relay connection status back to client process.
                printf("New Connection Request received at\nSource Port: %d, Destination Port: %d\n", reqPacket->source_port, reqPacket->dest_port);
                // Opening Write End for the port
                char fifoPath[50];
                sprintf(fifoPath, "%s%d.fifo", CLIENT_PORT_PATH, reqPacket->source_port);
                activeReq->portFd = Open(fifoPath, (O_EXCL | O_WRONLY));
                // Perform Handshaking
                int receivedFlag = hanshake(reqPacket->source_port, reqPacket->dest_port);
                printf("Received Packet: %d\n", receivedFlag);
                tcpClientPacket* toClient = (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
                if(receivedFlag == 0){
                    printf("Connection: EST\n");
                    activeReq->connectionStatus = EST;
                    toClient->info = EST;
                    toClient->isData = false;
                }
                else{
                    printf("Connection: RST\n");
                    activeReq->connectionStatus = RST;
                    toClient->isData = false;
                    toClient->info = RST;
                }
                printf("Client Packet sent back to client\n");
                print_tcpClientPacket(toClient);
                Write(activeReq->portFd, toClient, sizeof(tcpClientPacket));
                free(toClient);
            }
            else if(reqPacket->request == FIN){
                // TERMINATE Connection
            }
        }
        else{
            tcp_header* dataSegment = (tcp_header*)malloc(sizeof(tcp_header));
            if(activeReq->connectionStatus == EST){
                // Transfer Data
                if(reqPacket->isData){
                    dataSegment->source_port = reqPacket->source_port;
                    dataSegment->dest_port = reqPacket->dest_port;
                    memset(dataSegment->data, 0, BUFSIZ);
                    strcpy(dataSegment->data, reqPacket->data);
                    send_packet(network_fd[1], dataSegment, sizeof(tcp_header));
                    memset(dataSegment, 0, sizeof(tcp_header));
                }

                // Waiting for response
                tcpClientPacket* sendData = (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
                readBytes = receive_packet(network_fd[0], dataSegment, sizeof(tcp_header));
                memset(sendData->data, 0, BUFSIZ);
                if(readBytes > 20){
                    sendData->isData = true;
                    strcpy(sendData->data, dataSegment->data);
                    Write(activeReq->portFd, sendData, sizeof(tcpClientPacket));
                }
                free(sendData);
            }
            else if(activeReq->connectionStatus == CLOSED){
                // Closed Connection
            }
            else if(activeReq->connectionStatus == REQ){
                // Redirect for hanshake
            }
            free(dataSegment);
        }
```

```
    }
}

int main(){
    // For random sequence number generation in 'handshake' function.
    srand(time(NULL));

    // Creating two way 'Network' FIFO, emulating NETWORK
    // BOTH client TCP and server TCP will communicate through NETWORK
    twoWayFifo(network_fd, NETWORK_FIFO_PATH, O_EXCL, true);


    // Creating system fifo, all process must send requests to TCP via system.fifo
    while(true){
        printf("Waiting for Client Processes\n");
        // For client to TCP_client communication, we have 'system.fifo'
        // TCP will communicate via source port mentioned by client process in their request packets.
        system_fd = Open(SYSTEM_FIFO_PATH, (O_EXCL | O_RDONLY));
        // Will handle client only if there are active client process seeking connections
        handleClient();
    }


    // Closing System FIFO
    Close(system_fd);
    // Closing Network FIFO
    closeTwoWayFifo(network_fd);
    return 0;
}
```

Listing A.1: Client-side TCP Process

A.2 Server-side TCP Process

```
#include "tcp.h"

#define SERVER_PORT_PATH "serverSide/ports/"
#define SYSTEM_FIFO_PATH "serverSide/system.fifo"

// Record List
connectionList* head = NULL;

// 'Network' File Descriptor
int network_fd[2];
int system_fd;

bool checkPassiveOpen(connectionList* current){
    printf("Checking for Passive open at PORT: %d\n", current->dest_port);
    char fifoPath[50];
    sprintf(fifoPath, "%s%d.fifo", SERVER_PORT_PATH, current->dest_port);
    if( (current->portFd = Open(fifoPath, (O_EXCL | O_WRONLY))) == -1){
        if(errno = 2){
            // If no such file exist that implies that no server process has opened it.
            // Hence no server process is listening to PORT. Thus no passive open.
            printf("No Passive open found at PORT: %d\n", current->dest_port);
            return false;
        }
    }
    printf("File Descriptor: %d\n", current->portFd);
    printf("System File descriptor: %d\n", system_fd);
    // If exists then portFd has the Write file descriptor.
    printf("Found Passive Open at PORT: %d\n", current->dest_port);
    return true;
}

// Get connection history function
connectionList* getConnections(uint16_t sourcePort, uint16_t destinationPort){
    printf("Fetching Connection details\n");
    if(head == NULL){
        printf("Connection Table Empty...\n");
        head = (connectionList*)malloc(sizeof(connectionList));
        head->source_port = sourcePort;
        head->dest_port = destinationPort;
        head->connectionStatus = IDLE;
        return head;
    }
    else{
        connectionList* curr = head;
        connectionList* prev = NULL;
        while(curr){
            if(curr->source_port == sourcePort && curr->dest_port == destinationPort){
                return curr;
            }
            prev = curr;
            curr = curr->next;
        }
        // printf("New Connection\n");
        // Adding new node at last and returning it for the handler to update the details
        printf("No existing record found, creating new record\n");
        connectionList* newNode = (connectionList*)malloc(sizeof(connectionList));
        newNode->source_port = sourcePort;
        newNode->dest_port = destinationPort;
        newNode->connectionStatus = IDLE;
        prev->next = newNode;
        return newNode;
    }
}

// Performs Handshake for the given SYN PACKET
int handshake(tcp_header* syn_packet){
    printf("Performing Connection\n");
    tcp_header ack_syn_packet, ack_packet;
    // Register Connection
    // Add Sequence number, src_port, dest_port
    // Send ACK and SYN segment
    int server_seqnum = rand();
    ack_syn_packet = create_tcp_header(syn_packet->dest_port, syn_packet->source_port,
        server_seqnum, (syn_packet->sequence_num + 1), ACK_SYN_PACKET, 0, 0, 0);
    printf("Sending ACK+SYN packet\n");
    printTcpHeader(&ack_syn_packet);
    send_packet(network_fd[1], &ack_syn_packet, sizeof(ack_syn_packet));

    // Receive ACK segment for the SYN request
    // Open Connection.

    ssize_t receivedPacketSize = receive_packet(network_fd[0], &ack_packet,
        sizeof(ack_packet));
    if(receivedPacketSize != sizeof(ack_packet)){
        printf("Packet Lost");
        return 0;
    }
    if(ack_packet.data_offset_flags == ACK_PACKET){
        printf("Received ACK packet\n");
        printTcpHeader(&ack_packet);
        printf("ACK received for Connection with SEQ NUM: %d\n", ack_packet.sequence_num);
    }
    return 1;
}

void handleClientReq(void){
    printf("Handling Client\n");
    while(true){
        tcp_header* rxPacket = (tcp_header*)malloc(sizeof(tcp_header));
        // Listen for incoming requests. Blocking read operation on FIFO
        ssize_t receivedPacketSize = receive_packet(network_fd[0], rxPacket,
            sizeof(tcp_header));
        printf("Received:\n");
        printTcpHeader(rxPacket);
        // Checking for the record the connection.
        connectionList* currentConnection = getConnections(rxPacket->source_port,
            rxPacket->dest_port);
        printf("Connection Record Obtained...\n");
        print_connectionList(currentConnection);
        if(currentConnection->connectionStatus == IDLE){
            // Check for Passive open on the received Destination Port from Client Side.

            if(checkPassiveOpen(currentConnection) && rxPacket->data_offset_flags ==
                SYN_PACKET){
                // Proceed for Handshake
                if(handshake(rxPacket)){
                    // If ACK received for ACK+SYN packet, Open connection
                    printf("Connection Established for SRC: %d\t DST:
                        %d\n",currentConnection->source_port, currentConnection->dest_port);
                    currentConnection->connectionStatus = EST;
                    tcpClientPacket *serverPacket =
                        (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
                    serverPacket->isData = false;
                    serverPacket->info = EST;
                    // Sending Connection Packet to listening process.
                    printf("Informing the Server Application at PORT: %d\n",
                        currentConnection->dest_port);
                    print_tcpClientPacket(serverPacket);
                    Write(currentConnection->portFd, serverPacket, sizeof(tcpClientPacket));
                    free(serverPacket);
                }
                else{
                    printf("Unable to handshake.\n");
                    currentConnection->connectionStatus = IDLE;
                }
            }
            else{
                // Send RST PACKET
                printf("No Passive Open at PORT: %d, Sending RST Packet\n",
                    currentConnection->dest_port);
                currentConnection->connectionStatus = RST;
                tcp_header rst_packet;
                rst_packet = create_tcp_header(rxPacket->dest_port, rxPacket->source_port, 0,
                    rxPacket->sequence_num + 1, RST_PACKET, 0, 0, 0);
                send_packet(network_fd[1], &rst_packet, sizeof(tcp_header));
            }
        }
        else if(currentConnection->connectionStatus == EST){
            if(receivedPacketSize > 20){
                // Contains Data
                printf("Data Segment Received.\n");
                tcpClientPacket* serverPacket =
                    (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
                serverPacket->info = EST;
                serverPacket->isData = true;
                memset(serverPacket->data, 0, BUFSIZ);
                strcpy(serverPacket->data, rxPacket->data);
                print_tcpClientPacket(serverPacket);
                // Send Client Req
                printf("Relaying it to Server Application at %d\n",
                    currentConnection->dest_port);
                Write(currentConnection->portFd, serverPacket, sizeof(tcpClientPacket));

                memset(serverPacket, 0, sizeof(tcpClientPacket));

                // Get response
                printf("Waiting for Server response.....\n");
                Read(system_fd, serverPacket, sizeof(tcpClientPacket));

                tcp_header* dataSegment = (tcp_header*)malloc(sizeof(tcp_header));

                if(serverPacket->isData){
                    copyTcpHeader(dataSegment, rxPacket);
                    memset(dataSegment->data, 0, BUFSIZ);
                    strcpy(dataSegment->data, serverPacket->data);
                    printf("Sending the data back to client.\n");
                    send_packet(network_fd[1], dataSegment, sizeof(tcp_header));
                }
                free(dataSegment);
                free(serverPacket);
            }
            else if(rxPacket->data_offset_flags == FIN_PACKET){
                // Terminate Connection.
            }
        }
        free(rxPacket);
    }
}

int main(){
    srand(time(NULL));
    // Creating two way 'Network' FIFO, emulating NETWORK
    // BOTH client TCP and server TCP will communicate through NETWORK
    twoWayFifo(network_fd, NETWORK_FIFO_PATH, O_EXCL, false);

    while(true){
        printf("Waiting for Server Applications...\n");
        system_fd = Open(SYSTEM_FIFO_PATH, (O_EXCL | O_RDONLY));
        handleClientReq();
    }

    closeTwoWayFifo(network_fd);
    return 0;
}
```

Listing A.2: Server-side TCP Process

A.3 Client Application

```
#include "tcp.h"

#define SYS_FIFO_PATH "clientSide/system.fifo"
#define CLIENT_PORT_PATH "clientSide/ports/"
#define FILEREQ_PATH "test.txt"

void connect(uint16_t srcPort, uint16_t dest_port, connections req);
int sysFd;
int portFd;

int main(int argc, char *argv[]){


    sysFd = Open(SYS_FIFO_PATH, (O_EXCL | O_WRONLY));
    uint16_t sourcePort = atoi(argv[2]);
    uint16_t destPort = 80;
    // printf("I am able to open\n");
    connect(sourcePort, destPort, REQ);

    // Sending File Address
    clientReqPacket dataPacket;
    dataPacket.isData = true;
    dataPacket.source_port = sourcePort;
    dataPacket.dest_port = destPort;
    if(argc == 0){
        printf("Enter file path: ");
        fgets(dataPacket.data, BUFSIZ, stdin);
    }
    else{
        strcpy(dataPacket.data, argv[1]);
    }

    Write(sysFd, &dataPacket, sizeof(clientReqPacket));

    // File Address Sent

    // Receive Content
    printf("Waiting for response....\n");
    ssize_t readBytes;
    tcpClientPacket *data = (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
    readBytes = Read(portFd, data, sizeof(tcpClientPacket));

    if(data->isData == true){
        printf("Data Received\n");
        printf("%s\n", data->data);
    }

    // Closing PORT fifo
    Close(portFd);
    // Closing system.fifo
    Close(sysFd);

    return 0;
}

void connect(uint16_t srcPort, uint16_t dest_port, connections req){
    clientReqPacket ctrlPacket;

    ctrlPacket.source_port = srcPort;
    ctrlPacket.dest_port = dest_port;
    ctrlPacket.request = req;
    write(sysFd, &ctrlPacket, sizeof(ctrlPacket));

    char fifoPath[50];
    sprintf(fifoPath, "%s%d.fifo", CLIENT_PORT_PATH, srcPort);
    printf("Opening port....\n");
    portFd = Open(fifoPath,(O_EXCL | O_RDONLY));
    printf("File Opened...\n");
    ssize_t readBytes;
    tcpClientPacket rxPacket;
    readBytes = Read(portFd, &rxPacket, sizeof(tcpClientPacket));
    printf("Bytes Read: %ld\n", readBytes);
    print_tcpClientPacket(&rxPacket);
    // Receiving Control packet
    if(rxPacket.isData == false){
        if(rxPacket.info == EST){
            printf("Connection Established at Port %d\n", dest_port);
        }
        else{
            printf("No server application on Port: %d", dest_port);
            exit(EXIT_FAILURE);
        }
    }
}
```

Listing A.3: Client Process

A.4 Server Application

```
#include "tcp.h"

#define SERVER_PORT_PATH "serverSide/ports/"
#define SYSTEM_FIFO_PATH "serverSide/system.fifo"

int portFd;
int sysFd;

void listen(uint16_t destinationPort){
    // This shall keep on listening to PORT
    tcpClientPacket rxPacket;
    ssize_t readBytes;

    char fifoPath[50];
    sprintf(fifoPath, "%s%d.fifo", SERVER_PORT_PATH, destinationPort);

    portFd = Open(fifoPath, (O_EXCL | O_RDONLY));
    printf("Listening on Port %d.....\n", destinationPort);
    while( (readBytes = Read(portFd, &rxPacket, sizeof(tcpClientPacket))) > 0){
        print_tcpClientPacket(&rxPacket);
        if(rxPacket.info == EST){
            return;
        }
        else{
            printf("idk...\n");
        }
    }
}

int main(){

    sysFd = Open(SYSTEM_FIFO_PATH, (O_EXCL | O_WRONLY));

    printf("File Server is live....\n");
    uint16_t portNum = 80;
    while(true){
        listen(portNum);
        printf("Connected...\n");
        while(true){
            tcpClientPacket* dataPacket = (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
            ssize_t readBytes = Read(portFd, dataPacket, sizeof(tcpClientPacket));
            if(dataPacket->isData){
                printf("File Requested: %s\n", dataPacket->data);
                size_t len = strlen(dataPacket->data);

                if(dataPacket->data[len-1] == '\n'){
                    // Removing newline character from the path.
                    // Since fgets stops read at newline but buffers it.
                    dataPacket->data[len-1] = '\0';
                }
                int filefd;
                tcpClientPacket* txPacket = (tcpClientPacket*)malloc(sizeof(tcpClientPacket));
                txPacket->isData = true;
                memset(txPacket->data, 0, BUFSIZ);
                if( (filefd = open(dataPacket->data, O_RDONLY)) < 0){
                    // No such file exists
                    strcpy(txPacket->data, "No File Exist");
                    Write(sysFd, txPacket, sizeof(tcpClientPacket));
                }
                else{
                    // File exist, read file content and send
                    Read(filefd, txPacket->data, BUFSIZ);
                    Write(sysFd, txPacket, sizeof(tcpClientPacket));
                }
                free(txPacket);
                close(filefd);
            }
            else{
                if(dataPacket->info == FIN){
                    // Connection Termination
                    // Get out of this loop, no further file request from same client.
                    break;
                }
            }
        }
        close(portFd);
    }
    close(sysFd);
    return 0;
}
```

Listing A.4: Server Process

A.5 Header File

```
#ifndef TCP_H

#define TCP_H
// Libraries
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <stdint.h>
#include <time.h>
#include <semaphore.h>

#include <errno.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>

// Macros
#define NETWORK_FIFO_PATH "net/network"
#define FILE_MODE (S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH)
#define FIFO_RFLAG (O_RDONLY | O_NONBLOCK | O_EXCL)
#define FIFO_WFLAG (O_WRONLY | O_NONBLOCK | O_EXCL)


// TCP DATA OFFSET and FLAGS
#define SYN_PACKET ((5 << 12) | (1 << 1))
#define ACK_SYN_PACKET ((5 << 12) | (1 << 1) | (1 << 4))
#define ACK_PACKET ((5 << 12) | (1 << 4))
#define RST_PACKET ((5 << 12) | (1 << 2))
#define FIN_PACKET ((5 << 12 ) | (1))

// Structures
typedef enum {CLOSED, EST, FIN, LISTEN, REQ, RST, IDLE} connections;

// TCP Header Packet
typedef struct{
    uint16_t source_port;      // 16 bits: Source Port
    uint16_t dest_port;        // 16 bits: Destination Port
    uint32_t sequence_num;     // 32 bits: Sequence Number
    uint32_t ack_num;          // 32 bits: Acknowledgment Number
    uint16_t data_offset_flags; // 16 bits: Data Offset (4 bits) and Control Flags (9 bits)
    uint16_t window_size;      // 16 bits: Window Size
    uint16_t checksum;         // 16 bits: Checksum
    uint16_t urgent_ptr;       // 16 bits: Urgent Pointer
    char data[BUFSIZ];
} tcp_header;

typedef struct connectionList{
    uint16_t source_port;
    uint16_t dest_port;
    connections connectionStatus;
    int portFd;    // Write end file descriptor
    struct connectionList *next;
} connectionList;

typedef struct{
    uint16_t source_port;
    uint16_t dest_port;
    bool isData;
    connections request;
    char data[BUFSIZ];
} clientReqPacket;

typedef struct{
    char data[BUFSIZ];
    bool isData;
    connections info;
} tcpClientPacket;



// Print funcitons
void printTcpHeader(const tcp_header *header);
void print_connectionList(connectionList* node);
void print_tcpClientPacket(tcpClientPacket* packet);
void print_clientReqPacket(clientReqPacket* packet);




// Wrapper Functions
// Wrapper for FIFO

void Mkfifo(char* _path, mode_t _mode);

void twoWayFifo(int *fd, char* _path, int _flags, bool isClient);

void closeTwoWayFifo(int *fd);

void createTwoWayFifo(char* _path);

// Wrapper for pipe
void Pipe(int *fd);

void Close(int fd);

void Write(int fd, const void* buf, size_t nbytes);

int Open(char *_path, int _oflag);


ssize_t Read(int fd, void* buf, size_t nbytes);

pid_t Fork();

pid_t Waitpid(pid_t childPid, int *status_ptr, int options);


// Connection -> Handshake

tcp_header create_tcp_header(
    uint16_t source_port, uint16_t dest_port, uint32_t sequence_num,
    uint32_t ack_num, uint16_t data_offset_flags, uint16_t window_size,
    uint16_t checksum, uint16_t urgent_ptr);

void send_packet(int fd, tcp_header *packet, size_t packetSize);
ssize_t receive_packet(int fd, tcp_header *packet, size_t packetSize);
void copyTcpHeader(tcp_header *dest, const tcp_header *src);
#endif
```

Listing A.5: TCP.h

```
#include "tcp.h"


// Wrapper for pipe
void Pipe(int *fd){

    if(pipe(fd) == -1){
        perror("Error occured while creating Pipe");
        exit(EXIT_FAILURE);
    }
}

int Open(char *_path, int _oflag){
    int fd;

    if( (fd = open(_path, _oflag)) == -1){
        if(errno == 2 && ((_oflag & 1) == 0)){
            // If FILE does not exist and
            // attempt was made to Open for Read
            // Only then create FIFO
            Mkfifo(_path, FILE_MODE);
            // Again try to open it.
            return Open(_path, _oflag);
        }

        perror("Error opening file");
        printf("errno = %d\n", errno);
        exit(EXIT_FAILURE);
    }
    return fd;
}

void Close(int fd){
    if(close(fd) == -1){
        perror("Error occured while Close");
        exit(EXIT_FAILURE);
    }
}

void Write(int fd, const void* buf, size_t nbytes){
    if(write(fd, buf, nbytes) == -1){
        perror("Error occured while Writing");
        exit(EXIT_FAILURE);
```

```c
    }
}

void Mkfifo(char *_path, mode_t _mode){
    if(mkfifo(_path, _mode) == -1){
        perror("Error while creating FIFO");
        exit(EXIT_FAILURE);
    }
}

void createTwoWayFifo(char* _path){
    char readFilePath[20];
    char writeFilePath[20];

    strcpy(readFilePath, _path);
    strcpy(writeFilePath, _path);

    printf("File paths: %s %s\n",readFilePath, writeFilePath);
    strcat(readFilePath,".r");
    strcat(writeFilePath,".w");

    printf("File paths: %s %s\n",readFilePath, writeFilePath);

    // Creating FIFO
    Mkfifo(readFilePath, FILE_MODE);
    Mkfifo(writeFilePath, FILE_MODE);

    printf("Created FIFOs\n");
}

void twoWayFifo(int *fd, char* _path, int _flags, bool isClient){
    char readFilePath[20];
    char writeFilePath[20];

    strcpy(readFilePath, _path);
    strcpy(writeFilePath, _path);

    strcat(readFilePath,".r");
    strcat(writeFilePath,".w");
    // Creating Read-Write ends and returning the descriptors

    /*
    -> .r FIFO is client -> read and server -> write
    -> .w FIFO is client -> write and server -> read
    */
    if(isClient){
        fd[0] = Open(readFilePath, (_flags | O_RDONLY));
        fd[1] = Open(writeFilePath, (_flags | O_WRONLY));
    }
    else{
        fd[1] = Open(readFilePath, (_flags | O_WRONLY));
        fd[0] = Open(writeFilePath, (_flags | O_RDONLY));
    }
}
```

```
void closeTwoWayFifo(int *fd){
    Close(fd[0]);
    Close(fd[1]);
}

ssize_t Read(int fd, void* buf, size_t nbytes){
    ssize_t n;

    if((n = read(fd, buf, nbytes)) == -1){
        perror("Error occured while Reading");
        printf("Error No -> %d", errno);
        exit(EXIT_FAILURE);
    }

    return n;
}

// Wrapper for Processes
pid_t Fork(){
    pid_t pid;
    if( (pid = fork()) == -1){
        perror("Error occured while Fork");
        exit(EXIT_FAILURE);
    }
    return pid;
}

pid_t Waitpid(pid_t childPid, int *status_ptr, int options){
    pid_t returnId;

    if((returnId = (childPid, status_ptr, options)) == -1){
        perror("Error occured while Waitpid");
        exit(EXIT_FAILURE);
    }
    return returnId;
}



// TCP Connection -> 2-way handshake

tcp_header create_tcp_header(
    uint16_t source_port, uint16_t dest_port, uint32_t sequence_num,
    uint32_t ack_num, uint16_t data_offset_flags, uint16_t window_size,
    uint16_t checksum, uint16_t urgent_ptr) {
    tcp_header hdr;
    hdr.source_port = source_port;
    hdr.dest_port = dest_port;
    hdr.sequence_num = sequence_num;
    hdr.ack_num = ack_num;
    hdr.data_offset_flags = data_offset_flags;
    hdr.window_size = window_size;
    hdr.checksum = checksum;
    hdr.urgent_ptr = urgent_ptr;
    return hdr;
}

void send_packet(int fd, tcp_header *packet, size_t packetSize){
    Write(fd, packet, packetSize);
}

ssize_t receive_packet(int fd, tcp_header *packet, size_t packetSize){
    ssize_t receiveBytes;

    receiveBytes = Read(fd, packet, packetSize);

    return receiveBytes;
}

////////////////////////////////////////////////////////////

void printTcpHeader(const tcp_header *header) {
    printf("-------------------------------\n");
    printf("Source Port: %d\n", header->source_port);
    printf("Destination Port: %d\n", header->dest_port);
    printf("Sequence Number: %d\n", header->sequence_num);
    printf("Acknowledgment Number: %d\n", header->ack_num);
    printf("Data Offset and Control Flags: 0x%x\n", header->data_offset_flags);
    printf("Window Size: %u\n", header->window_size);
    printf("Checksum: 0x%d\n", header->checksum);
    printf("Urgent Pointer: %u\n", header->urgent_ptr);
    printf("-------------------------------\n");
}

void copyTcpHeader(tcp_header *dest, const tcp_header *src) {
    dest->source_port = src->source_port;
    dest->dest_port = src->dest_port;
    dest->sequence_num = src->sequence_num;
    dest->ack_num = src->ack_num;
    dest->data_offset_flags = src->data_offset_flags;
    dest->window_size = src->window_size;
    dest->checksum = src->checksum;
    dest->urgent_ptr = src->urgent_ptr;
}


// Print Functions
void print_connectionList(connectionList* node){
    printf("-----------------------------------\n");
    printf("Connection Record:\n");
    printf("Source Port: %d\n", node->source_port);
    printf("Destination Port: %d\n", node->dest_port);
    printf("port fd: %d\n",node->portFd);
    printf("Connection Request type: %d\n", node->connectionStatus);
    printf("-----------------------------------\n");
}

```

```
void print_tcpClientPacket(tcpClientPacket* packet){
    printf("-------------------------------\n");
    printf("Packet from Client_TCP\n");
    printf("isData: %d\n", packet->isData);
    printf("Connections Type: %d\n", packet->info);
    printf("Data -> %s\n", packet->data);
    printf("-------------------------------\n");
}

void print_clientReqPacket(clientReqPacket* packet){
    printf("-------------------------------\n");
    printf("Source Port: %d\n", packet->source_port);
    printf("Destination Port: %d\n", packet->dest_port);
    printf("Request: %d\n",packet->request);
    printf("-------------------------------\n");
}
```

Listing A.6: TCP.c